CMU-CS-78-114

A Production System Version of the Hearsay-II Speech Understanding System

Donald McCracken

April 1978

DEPARTMENT
of
COMPUTER SCIENCE

Carnegie-Mellon University

18 19 REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFOSR-TR- 78-1250	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) 6 A PRODUCTION SYSTEM VERSION OF THE HEARSAY-II SPEECH UNDERSTANDING SYSTEM.		5. TYPE OF REPORT & PERIOD COVERED 9 Interim Rept.
		6. PERFORMING ORG. REPORT NUMBER 14 CMU-CS-78-114
7. AUTHOR(s) 10 Donald McCracken		8. CONTRACT OR GRANT NUMBER(s) 15 F44620-73-C-0074, ARPA Order-2466
9. PERFORMING ORGANIZATION NAME AND ADDRESS Carnegie-Mellon University Department of Computer Science Pittsburgh, Pennsylvania 15213		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61101E A02466/7
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Blvd. Arlington, Virginia 22209		12. REPORT DATE 11 April 1978
		13. NUMBER OF PAGES 135
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Air Force Office of Scientific Research/NM Bolling AFB, Washington, DC 20332		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

12 137 p.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

16 2466 17 07

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A prime candidate organization for large, knowledge-rich systems is that of a production system (PS). PSs are rule-based architectures that have been used successfully for tasks ranging from models of human behavior to large application systems in chemistry and medicine, to classical artificial intelligence programs.

The question studied by this thesis is whether a PS architecture (PSA) helps or hinders with respect to implementation problems encountered by

20. Abstract

Hearsay-II (HSII), a large artificial intelligence system for understanding speech, developed at Carnegie-Mellon University (CMU). This is an important question because many of these problems, such as efficiency, compensating for error, controlling directionality, augmenting knowledge, and analyzing performance, have become limiting factors for performance.

To obtain an answer to this question, an actual system (call HSP, for "HearSay-Production system") was implemented on C.mmp, the CMU multi-miniprocessor, with a portion of the HSII speech knowledge translated into productions. An early decision was made to maintain close comparability of HSP with HSII rather than explore the more general question of how to best understand speech with a PS. Two knowledge-source (KS) programs from a complete HSII configuration were completely translated and run in HSP, and these provide a basis for some detailed comparisons between HSII and HSP. Ten other KSs were translated, and their static structure provides supporting evidence.

The HSP architecture was heavily influenced by HSII, itself similar to a PSA, and by a general PSA design philosophy manifested at CMU in systems such as PSG, PSNLST and OPS. HSP has several novel features when compared with these three relates PSAs, and thus makes a minor contribution in the area of PSA design.

The main results of the thesis are presented as a list of 17 assertions organized into five categories: Representation and Architecture, Space Efficiency, Time Efficiency, Parallelism, and the Small Address Problem. The HSP architecture is found to be adequate for representing the HSII speech knowledge, even though HSP is simple compared to other PSAs.

Space and time efficiency are another matter. There is a moderate space penalty for representing declarative HSII knowledge as HSP productions, which is cause for concern since HSII contains many large declarative knowledge structures. Even more serious is the substantial space inefficiency of the global HSP working memory, since it must be used in place of large, highly optimized local working memories typically used by HSII KSs.

HSPs lack of local working memory results also in a large loss of time efficiency because of heavier use of data-directed control and greater creation/read/write costs in its global Working Memory. In the two-KS configuration this loss is a factor somewhere in the range 6 to 36, but projecting to a full KS configuration yields a much larger factor of 100 to 3000 since many of the KSs to be added make heavy use of local working memory and control (in their HSII form).

Some of the time efficiency handicap is made up through increased parallelism of HSP over HSII. A source of parallelism not exploited by HSII, called intra-KS parallelism, results from HSP's smaller knowledge unit size. We estimate conservatively a half order of magnitude increase in parallelism for a full KS configuration. It could be much greater than that if HSP's less powerful synchronization mechanisms turn out to be adequate with a full complement of KSs.

Finally, HSP is found to aid solution of the Small Address Problem, as it exists on C.mmp, by making it easy to do overlaying of both long-term knowledge and working memory.

The thesis concludes with brief discussion of 9 important questions which have emerged from the current study -- questions which must be answered to complete the evaluation of a PSA for HSII.

CMU-CS-78-114

2

A Production System Version of the Hearsay-II Speech Understanding System

Donald McCracken

April 1978

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pa. 15213

Submitted to Carnegie-Mellon University in partial fulfillment of the requirements for the degree of Doctor of Philosophy

This research was supported by the Defense Advanced Research Projects Agency under contract no. F44620-73-C-0074 and is monitored by the Air Force Office of Scientific Research.

Abstract

A prime candidate organization for large, knowledge-rich systems is that of a production system (PS). PSs are rule-based architectures that have been used successfully for tasks ranging from models of human behavior to large application systems in chemistry and medicine, to classical artificial intelligence programs.

The question studied by this thesis is whether a PS architecture (PSA) helps or hinders with respect to implementation problems encountered by Hearsay-II (HSII), a large artificial intelligence system for understanding speech, developed at Carnegie-Mellon University (CMU). This is an important question because many of these problems, such as efficiency, compensating for error, controlling directionality, augmenting knowledge, and analyzing performance, have become limiting factors for performance.

To obtain an answer to this question, an actual system (called HSP, for "HearSay-Production system") was implemented on C.mmp, the CMU multi-miniprocessor, with a portion of the HSII speech knowledge translated into productions. An early decision was made to maintain close comparability of HSP with HSII rather than explore the more general question of how to best understand speech with a PS. Two knowledge-source (KS) programs from a complete HSII configuration were completely translated and run in HSP, and these provide a basis for some detailed comparisons between HSII and HSP. Ten other KSs were translated, and their static structure provides supporting evidence.

The HSP architecture was heavily influenced by HSII, itself similar to a PSA, and by a general PSA design philosophy manifested at CMU in systems such as PSG, PSNLST and OPS. HSP has several novel features when compared with these three related PSAs, and thus makes a minor contribution in the area of PSA design.

The main results of the thesis are presented as a list of 17 assertions organized into five categories: Representation and Architecture, Space Efficiency, Time Efficiency, Parallelism, and the Small Address Problem. The HSP architecture is found to be adequate for representing the HSII speech knowledge, even though HSP is simple compared to other PSAs.

Space and time efficiency are another matter. There is a moderate space penalty for representing declarative HSII knowledge as HSP productions, which is cause for concern since HSII contains many large declarative knowledge structures. Even more serious is the substantial space inefficiency of the global HSP working memory, since it must be used in place of large, highly optimized local working memories typically used by HSII KSs.

HSP's lack of local working memory results also in a large loss of time efficiency because of heavier use of data-directed control and greater creation/read/write costs in its global Working Memory. In the two-KS configuration this loss is a factor somewhere in the range 6 to 36, but projecting to a full KS configuration yields a much larger factor of 100 to 3000 since many of the KSs to be added make heavy use of local working memory and control (in their HSII form).

Some of the time efficiency handicap is made up through increased parallelism of HSP over HSII. A source of parallelism not exploited by HSII, called intra-KS parallelism, results from HSP's smaller knowledge unit size. We estimate conservatively a half order of magnitude increase in parallelism for a full KS configuration. It could be much greater than that if HSP's less powerful synchronization mechanisms turn out to be adequate with a full complement of KSs.

Finally, HSP is found to aid solution of the Small Address Problem, as it exists on C.mmp, by making it easy to do overlaying of both long-term knowledge and working memory.

The thesis concludes with brief discussion of 9 important questions which have emerged from the current study -- questions which must be answered to complete the evaluation of a PSA for HSII.

Table of Contents

Acknowledgements

To Allen Newell, my principal advisor, for many years of patient and understanding guidance.

To committee members Lee Erman and Victor Lesser for moral support and many fruitful discussions of Hearsay-II.

To David Klahr for his contribution of time and interest as the "external" committee member.

To the Hearsay-II knowledge-source implementers, especially Rich Smith and Jack Mostow, for aid in understanding their creations.

To Raj Reddy and all members of the CMU Speech Group for providing a stimulating environment to give my research direction and meaning.

To the computer operations staff for endless hours of worry-free computer usage.

To the C.mmp/Hydra project for adding spice to my life and strength to my character.

To longtime compatriot George Robertson for his support in many ways.

And finally to Marilyn and Stephen for the will to carry on.

Chapter 1

Introduction

A prime candidate organization for large knowledge-rich systems is that of a production system (PS) [Newell, 1973; Davis and King, 1975; Waterman and Hayes-Roth, 1973]. PSs are rule-based architectures that have been used successfully for tasks ranging from models of human behavior [Newell and Simon, 1972],[1] to large application systems in chemistry [Feigenbaum, Buchanan and Lederberg, 1971] and medicine [Shortliffe, 1974], to general artificial intelligence programming tasks [Rychener, 1976]. The uniform rule structure of knowledge in PSs makes them particularly useful for tasks requiring iterative upgrading of knowledge content.

The question studied by this thesis is whether a PS architecture (PSA) helps or hinders with respect to problems encountered by Hearsay-II (HSII), a large artificial intelligence system for understanding speech, developed at Carnegie-Mellon University [CMU Speech Group, 1977; Lesser and Erman, 1977; Erman and Lesser, 1975; Lesser, Fennell, Erman and Reddy, 1975]. This is an important question because many of these problems, such as efficiency, compensating for error, controlling directionality, augmenting knowledge, and analyzing performance, have become limiting factors. Alleviation of one or more of these might clear the way for significantly better performance.

To obtain an answer to this question, an actual system (called HSP, for "HearSay-Production system") was implemented on C.mmp, the CMU multi-miniprocessor [Wulf and Bell, 1972], with a portion of the HSII speech knowledge translated into productions. Two knowledge-source (KS) programs from a full HSII configuration were completely translated and run in HSP, and these provide a basis for some detailed comparisons between HSII and HSP. Ten other KSs were translated, but with some simplifications and omissions, and were never completely debugged and run. However, their static structure does provide evidence for representation issues.

One promising remedy for the slowness of a system like HSII is the use of parallel machine architectures, and so HSP was designed with parallelism in mind. Data on processor utilization and overhead were obtained for HSP runs of the two-KS configuration with up to ten processors on C.mmp, and the results compared with a multiprocessor simulation of HSII [Fennell and Lesser, 1977]. In addition, an HSP simulator was built to work from timings collected by HSP in uniprocess mode and predict processor utilization with larger numbers of processors. The simulator was validated by comparing with real HSP multiprocessor runs.

This thesis also aims to make contributions in the area of PSA design, since HSP has

[1] Other examples are [Newell, 1973; Klahr, 1973; Young, 1973; Moran, 1973; Farley, 1974; Waterman, 1974; Brooks, 1975]. See [Rychener, 1976] for a brief discussion of each.

several novel features related to multiprocessing and the use of multiple, independent sources of knowledge.

An early choice point in the research was whether to make HSP mimic HSII, or to break free from HSII entirely. The determination to maintain comparability of HSP with HSII led to the former, yielding an imitative style of study. This precluded many directions that might have been followed if the focus had been on how best to understand speech with a PS, for example, studies of knowledge augmentation or directionality.[2] A result of this decision is that HSP fared badly in the comparison, though on the positive side we have shed light on representation and efficiency issues in both HSP and HSII.

From the beginning it was recognized that an effort of the scope of this thesis had a large number of associated risks:

> This was a one-man effort, whereas HSII is the product of several tens of man-years. Large amounts of HSII kernel and knowledge-source code had to be penetrated and thoroughly understood to accomplish the translation.
>
> HSII was undergoing rapid development during the period of the thesis work, so that the foundation for translation was continually crumbling away. A good deal of effort was wasted in trying to track the moving target, and in the end this was given up as futile. Thus of the 12 KSs translated for HSP, only 5 still existed in the final HSII version of September 1976.
>
> The risk which turned out to be most damaging was the immaturity of the underlying hardware and software: C.mmp [Wulf and Bell, 1972] and its operating system Hydra [Wulf et al., 1974]. These were still being actively developed, and the steady influx of new hardware and software took a large toll (even larger than anticipated) in system crashes and incompatibilities.

Even in the face of such risks, it was felt the problem had sufficient interest to be worth pursuing, with an understanding that the final results would inevitably focus on a small number of the total set of issues. The remainder of the introduction summarizes the major components of this study. First comes a characterization of the speech understanding task, followed by overviews of the two architectures: PS and HSII. The final component includes the underlying C.mmp hardware, operating system, and implementation system used by HSP.

[2] For this reason, it is improper to identify HSP as a "PS for speech understanding".

1.1 The Speech Understanding Task

The origin of much of the current activity in building speech-understanding systems dates back to the spring of 1970, when a study group was set up by the Advance Research Projects Agency of the Department of Defense. Its task was to consider the feasibility of developing a system that would understand human speech in the context of performing some task. The final report of that study group [Newell et al., 1973] concluded that a five-year development program to produce a research prototype system would have a reasonable chance of success, and a big payoff for the technical community at large. The term "Speech Understanding System" (SUS) was coined to distinguish the goal of understanding the intent of an utterance in the framework of some task from the (harder) goal of "recognition" of the utterance as consisting of particular words, phonemes, etc. The report laid down a set of specifications for the system along 19 dimensions which defined fairly precisely what the goals of the effort should be, and how success or failure could be measured.

The results of the five-year program are now in. It was successful not only in achieving the specific goals, but also in producing a large collection of scientific and technical advances [Medress et al., 1977; CMU Speech Group, 1977; Walker, 1976; Woods et al., 1976; Bernstein, 1976]. The system which succeeded (on 13 August 1976) is called HARPY [Lowerre, 1976], and was developed at CMU as an alternative to HSII (the system studied by this thesis). HARPY's performance was 91% sentence accuracy (95% semantic accuracy) on a 1011-word vocabulary with 5 speakers. HSII came very close to the goals with about 70% sentence accuracy (91% semantic accuracy) for a single speaker.

Several aspects of the speech understanding task make it stand out from many other artificial intelligence tasks:

- The rich structure of hierarchical information levels (phonetic, syllabic, lexical, etc.) and knowledge sources. This brings to the fore the interesting problems of knowledge interaction (conflict) and directionality control.

- The high degree of formalization of each level, with lexicons of basic entities and rules relating entities (both within and between levels). A consequence of this is that much existing speech knowledge is conveniently representable as simple tables (e.g., probability matrices and dictionaries) or networks (e.g., grammars and state transition networks).

- The large direct recognition component of speech understanding; i.e., little high-level serial reasoning.[3]

The ubiquity of error. Errors begin with the speech input itself; i.e., the variability due to noise and speaker. Imperfect theory leads to additional errors being introduced by inaccurate or incomplete knowledge-sources. As these errors propagate through the various hierarchical levels, they are compounded. The presence of error must be compensated by maintaining everywhere several plausible alternatives, and this results in turn in a serious combinatorial search problem

The Implementation Problems

The above aspects, and others shared with other artificial intelligence tasks, give rise to characteristic implementation problems that must be attended to by builders of SUSs. These are listed below in Figure 1.1;[4] additional qualification or listing of significant subproblems is included where appropriate. Note that problem 4 is more specific than the others. It was included because of its particular importance for the C.mmp architecture which underlies HSP.

[3] This is indicated by the fact that humans understand speech in real time. Information processing models of human problem solving [Newell and Simon, 1972] indicate a basic recognize-act cycle time of 20-100 milliseconds for the human serial processor, or only about 10-50 cycles per second. Thus, such a processor could not understand speech unless a great deal of the task were accomplished by direct recognition. However, humans may have some special pipe-lining or other similar mechanisms to provide more effective parallelism for speech understanding.

[4] An important source for the construction of this problem list was the paper by Moore and Newell [1974] on the Merlin system, which contains a list of design features for the creation of an intelligent system.

1. Adequacy of representation
   Not mere possibility, but some degree of facility

2. Space efficiency
   Of both long-term and working memory

3. Time efficiency
   Real-time input
   Searching in combinatorial spaces
   Exploiting parallelism

4. Limited Address Space
   Large systems on small-address-space machines
   Also known as the Small Address Problem

5. Error
   In input
   In knowledge-sources

6. Directionality
   Integrating the activity of multiple idiosyncratic knowledge-sources

7. Augmentation
   Knowledge interaction problem

8. Testing

9. Debugging

10. Performance analysis

Figure 1.1 Characteristic implementation problems for Speech Understanding Systems

1.2 The Promise of Production Systems

A production system architecture (PSA), in the sense of this thesis, consists of a set of rules, called productions, stored in a production memory (PM), and a working memory (WM) which holds symbolic structures operated on by the productions. A production consists of a condition part which tests the current state of WM, and an action part which specifies additions/modifications/deletions to WM to take place in case the condition part is true. The basic operating cycle of a PS is a "recognize-act" cycle: i.e., recognize which production conditions are true of the current WM, then execute the corresponding production actions.[5] Since each cycle makes some change to the WM state (which changes the set of productions that are true), iteration of the cycle produces (potentially, at least) an interesting stream of behavior.

PSAs at large show a great deal of diversity. The strain to which HSP belongs has a lineage beginning with studies of human problem solving [Newell and Simon, 1972] and a PSA called PSG [Newell and McDermott, 1975], and extending to the more recent architectures of PSNLST [Rychener, 1976] and OPS [Forgy and McDermott, 1977]. A variant strain with roots in performance-oriented, knowledge-based expert systems (DENDRAL [Feigenbaum, Buchanan and Lederberg, 1971] and MYCIN [Shortliffe, 1974]) embodies a major architectural difference in the use of rules. These systems treat rules as declarative knowledge structures to be interpreted by a simple, inflexible goal-directed control mechanism;[6] i.e., the production rules are only a piece of the total system (though the most important piece). On the other hand, PSG-like systems have no declarative long-term knowledge; all knowledge is encoded as (procedural) productions, which combine knowledge with how it is to be used (control information).[7] The tradeoff between the two views is essentially this: PSG-like systems have more flexibility in representing control information (i.e., knowledge interactions), but at the cost of greater difficulties than MYCIN-like systems in augmentation and permitting multiple use of knowledge.

How a PSA Faces the Implementation Problems

The substantial promise that PSAs show for large artificial intelligence systems (specifically, speech understanding systems) can be made concrete by examining each of the problems introduced above. Unfortunately, as we shall see below, a PSA does not necessarily offer promise for all the problems; some may be exacerbated.

> Adequacy of representation -- The simplicity of PSA data and control representation is grounds for some doubt about adequacy. Medium-scale PS

[5] This is analogous to the "fetch-execute" cycle for sequential machines.

[6] Recent work by Davis [1976, Chapter 7] has shown how to incorporate meta-rules into MYCIN for strategies of knowledge use. This solution is distinctly different from the single-level rules of PSG-like systems.

[7] Winograd's [1975] discussion of declarative vs. procedural representations is relevant here.

efforts [Rychener, 1976] have met with success, but we must wait for evidence from large, complex systems; e.g., the Instructable Production System [Rychener and Newell, 1978].

Space efficiency -- The space efficiency of PSs is not yet clear. Rychener [1976] observed that his PSs seem to be less space efficient (a factor of three in one case) than the equivalent versions in standard artificial intelligence languages. As will be seen in Chapter 3, the large, simple knowledge tables of the speech task cause space problems for HSP.

Time efficiency -- The recognize-act cycle, with its inherently large ratio of recognition to action, matches well with the large recognition component of the speech task. However, the theoretically limitless recognition power of a PSA is just a dream on current machine architectures. This includes parallel architectures, since only tens of processors are typically available, while thousands may be needed.

Limited Address Space -- The decomposition of both long and short-term memory into structurally independent units (productions and WM elements, respectively) offers the possibility of overlaying these units in the address space. However, current mechanisms for efficient condition evaluation require additional large data structures which may be more difficult to overlay.

Error -- The recognition structure of PSs allows those errors which are detectable by direct recognition of the WM state (hopefully the majority) to be continuously checked, and (theoretically) at little cost to the main processing. The response to an error state could be the application of some specialized knowledge to correct the error and thus effect automatic recovery. If the PSA is one in which new information augments the WM rather than replacing information already there, this provides a sort of automatic history mechanism. Such a mechanism could be useful for reconstructing the reasons for an error.

Directionality -- Since PSs operate on a basis of immediate action through recognition, they will always attempt to apply every bit of knowledge which bears (directly) on the current situation. Thus the main problem for directionality becomes one of restricting potential behavior.[8] An obvious way to resolve a conflict among a set of productions is to make their conditions more discriminating by adding condition elements. These elements either test more detail of the natural WM state or test for special control signals in WM.

Augmentation -- The simple model of augmentation by additive growth is perhaps the most seductive feature of PSs, although a deceptive one. There remains the problem (as yet poorly understood) of how to achieve coordinated action between original and added productions.

[8] Although in cases where the desired direction of behavior is known in advance, there is certainly no problem in simply programming the PS to behave appropriately.

Testing -- There is a potential for rapid setup of test situations within a PS simply because there is no long-term processing state contained anywhere except in the WM. Thus many tests can be conducted by initializing WM to contain elements which will induce the behavior to be tested, and then just starting the PS. This is easy provided a modest number of elements suffices.

Debugging -- Monitors on the WM state can be written as normal productions and intermixed freely with task productions. Secondly, since the basic operation of the PS is interpretive, the control points already exist for adding facilities such as tracing and breakpoint. Finally, productions can easily be disabled temporarily to eliminate extraneous behavior when zeroing in on a problem.

Performance analysis -- The uniform structure of small, relatively independent productions, each with a definite function, seems to provide a fertile field for performance analysis. Ablation studies (i.e., observing the effect of selective removal of productions) should be possible in great variety because of the large number of productions. It should also be possible to trace the contributions of knowledge in great detail at the level of individual productions. Finally, an analysis of variance study (with individual productions or groups thereof taken as the factors in the analysis) might be useful in helping determine how productions interact, in both useful and detrimental ways.

The overall picture painted above is a rosy one indeed. A note of realism is in order: it is unlikely that all, or even most, of the positive promises will come to fruition. Some of them are highly speculative. As each aspect is better understood, there will be additional qualifications. And, as always in complex systems, problems which we think have been solved have often only moved to a different place. Thus, the above list must be considered a provisional one, serving only to guide the next wave of research.

1.3 Overview of the HSII Architecture

Hearsay-II (HSII) is a system organization for the speech understanding task designed to coordinate the contributions of diverse, essentially independent sources of knowledge which operate at the different hierarchical levels of speech knowledge.

The essential device for communication among knowledge sources (KSs) is a large shared working memory called the Blackboard (BB), which all KSs inspect and modify. The BB is a medium for growing linked networks of hypotheses (hyps) about what elements exist at the various levels and for various time intervals of the utterance (e.g., words, syllables, etc.). The BB levels form a hierarchy based on the time grain of the hypothesized elements at each level. Elements at the top level span the entire utterance, while the bottommost level contains elements corresponding to small segments of the utterance. A crucial feature of the BB is its capacity for representing and interrelating alternative hypotheses at all levels and time regions.

A uniform hyp structure is used at all levels of the BB, except for the fact that a different lexicon of elements exists at each level. A hyp is an attribute-value structure (the attributes are called fields) with entries for: the lexicon of the element, the element's identity, the element begin and end time within the utterance, links from supporting hyps at the next lower level, links to supported hyps at the next higher level, a current estimate of validity, and several other items. It is possible to particularize the hyp structure at any level by dynamically adding new fields and associated values which are relevant to KSs operating at that level; and this is an important feature that gets much usage. In addition to hyps, the BB contains links which consist of: a reference to the lower (supporting) hyp, a reference to the upper (supported) hyp, and an implication value indicating the strength of the support.

Upward growth in the BB normally consists of linking a hyp or a time-adjacent sequence of hyps at some level to a new hyp at the next higher level; this is essentially a recognition process, and the lower hyps are said to be supporting the upper one. Downward growth in the BB normally consists of hypothesizing and linking to a hyp or sequence of hyps at the next lower level; this is essentially a prediction process. Also, sideways growth may occur in the form of predictions of hyps in the context (right or left in utterance-time) of already existing hyps.

All the static speech knowledge in the system is encoded as procedural units (the KSs) written in SAIL, a compiler based, higher-level algebraic language [Reiser, 1976]. The KSs are structured to contribute their knowledge by creating and modifying BB hyps and links in response to relevant changes by other KSs (i.e., in a data-directed fashion). KSs are typically experts about the relationship between units at two adjacent levels in the BB, and work only locally in that area.[9] Every KS can be completely independent of all others except those that deal with the same BB levels it does, and even then communication is restricted to occur only through the structures in the BB.[10]

An initial portion of the condition for KS action is encoded in a separate procedural unit called a Precondition (PRE). Every KS has a corresponding PRE, although a single PRE commonly acts for more than one KS. A PRE is executed (to evaluate its condition) whenever a BB change occurs which belongs to a class which the PRE is interested in. If the PRE finds what it is looking for in the BB (evaluates to True), it instantiates the appropriate KSs to act on that local area of the BB. The main reason for having PREs separate from the KSs is to obtain enough preliminary context information about KS instantiations to allow their desirability to be estimated and used by the scheduler. Another reason for having PREs is that KS instantiation overhead is avoided when the PRE evaluates to False. (PREs remain in existence throughout an entire run, and thus there is little overhead in invoking one.)

The global behavior of the HSII system can be viewed as a heuristic search through

[9] However some KSs may deal with non-adjacent levels, and this freedom is a critical aspect of the architecture.

[10] This is the philosophy. The real system violates this in a number of places, usually for efficiency reasons.

the space of partial recognition networks being built up in the BB; these structures are in the form of hyps with validities, and validity-propagating links between them. The overall goal is to find a sufficiently valid network which spans all levels of the BB and the entire utterance time. The KSs expand the networks both bottom-up and top-down until they can be joined. Validities are periodically updated for the growing partial networks, and the validities also provide heuristics for the network-growing activity by indicating how and where the growing should proceed.

How HSII Faces the Implementation Problems

The HSII architecture was of course designed with the system problems of Figure 1.1 in mind, and hence has many characteristics intended to deal with the problems. The following list describes the most important of these characteristics:

> Adequacy of representation -- HSII plays it safe on adequacy for long-term knowledge by using SAIL, a general higher-level language, so that there is very little constraint on how knowledge is encoded in the KSs. As for working memory, SAIL data structures are adequate for KS-local use, and for global use the hyps and links of the BB can have arbitrary attribute-value structures appended.
>
> Time/space efficiency -- HSII KSs are individually "tailored" for efficiency, with specialized local data structures which save space and access time compared to their global (BB) equivalents. These local structures also avoid the substantial overheads of data-directed invocation which go with global BB use. Furthermore, the HSII architecture allows parallel execution of KS instantiations along three dimensions: different KSs, different utterance-time intervals, and alternative hypotheses. A parallel machine simulation study made within HSII [Fennell, 1975; Fennell and Lesser, 1977]) yielded an effective parallelism measure between 4 and 6, based on an earlier version of HSII which had a limited set of KSs. With a richer set of KSs and/or a reduction (or elimination) of synchronization overheads, the effective parallelism could be expected to increase dramatically.
>
> Limited address space -- The independence of KS modules permits them to be overlayed in address space (in this case swapped in from secondary memory).
>
> Error -- The main way in which HSII handles error is by tolerating it -- by making the knowledge ignore minor mismatches and inconsistencies (as for time adjacency tests), and by having alternative sources of knowledge. A crucial part of this is the representation in the BB of alternative (conflicting) hypotheses and their interrelationships.
>
> Directionality -- A mechanism within the HSII kernel schedules potential PRE and KS instantiations according to their "desirability", which is a function of the set of BB elements the KS or PRE is likely to manipulate, and of the current global state of the BB. In addition, individual KSs commonly use rating thresholds to limit their activity.
>
> Augmentation -- Normal program modification to the SAIL module for a KS is the

standard means of knowledge augmentation. However, most KSs are table or rule-driven and thus can be augmented in certain ways by simply adding table entries or rules. It is possible (and easy) to augment by integrating a new KS into the system (though implementing the KS itself may be hard). This is not often done, but has been instrumental in several major iterations of the system.

Testing -- Testing can be done reasonably effectively by operating with a minimal system configuration (i.e., the KS to be tested in a minimal environment), and by including in the KS code some special facilities for controlling test runs.

Debugging -- HSII has its own interactive command language and debugging tools, taking up a third of the system (exclusive of KSs), and representing a significant development effort in the early years of the project. These tools span all levels of system structure, from inner details to global interactions of KSs.

Performance analysis -- Much analysis is done on individual KSs running in minimal configurations. (The independence of KSs is what allows these configurations to work.) Causality analysis [Newell, 1975] (i.e., looking at which KSs make which changes, and why) carries most of the burden of performance analysis; the display facilities to support this require 11% of the system size. Ablation studies [Newell, 1975] are almost impossible in practice at the level of whole KSs due to excessive leanness of knowledge in the system (i.e., the existing KSs barely span all levels of the BB). However, individual KSs were coded to allow certain subparts to be switched on and off for evaluation purposes.

To summarize, HSII has devised means of dealing with these problems -- its success provides an independent demonstration of that. But the problems have not been completely solved, and many of them are limiting factors in attempts to improve performance.

1.4 The Systems Underlying HSP

All three components of the underlying system used to implement HSP (the C.mmp hardware, the Hydra operating system, and the L* implementation system) are unusual. Though they did have a strong impact on the HSP implementation effort itself, their effect for the most part did not carry through to the HSP architecture. If fact, HSP purposely avoids using operating system facilities wherever possible so that their effect need not be factored out of the data. Thus, no description of Hydra is necessary here. Interested readers may consult [Wulf et al., 1974].

The C.mmp hardware [Wulf and Bell, 1972] consists of 16 slightly modified PDP11 processors (currently 11 PDP11/40s and 5 PDP11/20s) connected to 16 primary memory modules via a 16 x 16 crosspoint switch, which allows each processor to access all the memory with only minor switch delay. Each processor has a set of relocation registers which map addresses at the processor to physical memory addresses. This hardware architecture does have a significant effect on HSP in at least two ways. First, and most obvious, is the potential for up to 16-way parallelism in program execution. HSP was designed with this potential fully in mind, and Chapter 5 reports the results on HSP's use of parallelism. A second effect is the uncomfortably small 16-bit address space of the processors, which provides a movable window on the large primary memory (about a million words). HSP incurred very large development costs, and somewhat smaller runtime costs, in dealing with this so-called Small Address Problem. Chapter 6 presents the details of this.

The principal impact of L* [Newell, McCracken and Robertson, 1977] on HSP is in efficiency, both of time and space. The central programming language of L* is interpretive, and although translation to machine code is possible, HSP was left entirely interpretive. Inefficiency of space comes from the encoding of data structures as L* list structures. These effects must be factored out when time and space comparisons are made with HSII. The substantial benefits of L* came during HSP development: flexibility to allow rapid iteration and experimentation, plus an overlay facility to help with the small address problem.

1.5 Organization of the Thesis

The body of the thesis has the following organization: Chapter 2 introduces the HSP architecture and contrasts it with several related PSAs developed at CMU. The intermediate chapters (3 through 6) deal with the main results of the thesis, covering representation (Chapter 3), time efficiency (Chapter 4), parallelism (Chapter 5), and the small address problem (Chapter 6). The conclusion (Chapter 7) presents a complete recapitulation of the results, structured as a list of assertions followed by a list of important unanswered questions. The intermediate chapters contain no significant conclusions that are not reflected in Chapter 7. They do contain background and details of how the evidence was developed. No separate summaries are included in intermediate chapters; the conclusion chapter serves that purpose since its organization closely follows the division of the intermediate chapters.

Chapters 1 and 7 form a nearly self-contained unit, so that a useful strategy for reading the thesis is to concentrate on those two, and merely skim the others for background and a flavor of the details.

For the sake of completeness, detailed specifications of the HSP architecture appear in Appendix A. This will not be relevant to the general reader. Potentially more valuable is Appendix B, which contains annotated sample HSP productions. These can be referred to for a quick overall impression of HSP.

Chapter 2

The HSP Architecture

There were two major driving forces behind the design of the architecture for the HSP system. The first was the nature of the speech task itself and the model provided by the HSII architecture, which is similar in many ways to a PSA [Fennell, 1975]. The second was a general PSA design philosophy, manifested in systems developed at CMU such as PSG [Newell, 1972; Newell and McDermott, 1975], PSNLST [Rychener, 1976] and OPS [Forgy and McDermott, 1977].[1] These three architectures represent a historical progression in PSA development at CMU (PSG: 1971, PSNLST: 1974, OPS: 1976); HSP is a contemporary of OPS. Comparisons between HSP and other PSAs will be restricted for the most part to just those three systems; the close kinship of the systems makes the comparisons sharper, and will not take us too far afield.[2]

There is one other related PSA which should be mentioned: the one used for the SASS module in HSII [Mostow and Hayes-Roth, 1978]. Its productions use condition and action primitives specialized to its task of syntactic and semantic recognition (e.g., predicates such as ADJACENT, and actions such as CONCATENATE). They are at a grosser level of representation than HSP and its three kindred PSAs; typical actions involve SAIL procedures on the order of a hundred statements, rather than primitive modifications to a working memory. Also, the productions do not encode high-level control knowledge, that being the domain of the HSII focussing mechanism. Efficient evaluation of the productions is obtained with an Automatically Compilable Recognition Network (ACORN) [Hayes-Roth and Mostow, 1975]. The ACORN reduces redundancy of condition evaluation in much the same way as the mechanisms in OPS [Forgy, 1977].

This chapter has a double purpose: to briefly introduce the HSP architecture, and to compare it with PSG,[3] PSNLST and OPS. The features for comparison were chosen because of their importance in the overall shaping of HSP, or because they strongly distinguish HSP from the comparison systems. Figure 2.1 summarizes the comparisons. The remainder of this chapter elaborates on these, and in so doing also introduces the most important features of the HSP architecture. For more details on the architecture, consult the detailed HSP specifications in Appendix A, or the annotated productions in Appendix B.

---

[1] PSG was developed by Newell principally for building and testing models of human cognition; PSNLST was developed by Rychener for a study of the use of a PSA for a set of classic artificial intelligence tasks; OPS was developed for the Instructable PS project [Rychener and Newell, 1978] by Forgy, McDermott, Newell and Rychener.

[2] Wider-ranging PS comparisons can be found in [Davis and King, 1975].

[3] PSG actually represents a large family of of PSAs since many key features are parameterized. We assume here the default settings.

	PSG	PSNLST	OPS	HSP
Production Memory				
Add productions dynamically	Yes	Yes	Yes	No
Working Memory				
Simple list structure for elements	Yes	Yes	Yes	No
Arbitrary nesting	Yes	No	Yes	No
Explicit WM element references	No	No	No	Yes
Conditions				
Explicit condition on change	No	No	No	Yes
Arbitrary predicates	No	Yes	Yes	No
Match arbitrary WM element subpart	No	No	No	Yes
Disjunctions	Yes	Yes	Yes	No
Negated conjunctions	No	Yes	Yes	No
Bind variable to whole WM element	No	No	Yes	Yes
Actions				
Modify WM element subfields	Yes	No	No	Yes
Control				
Conflict resolution	Yes	Yes	Yes	No
Fire multiple productions per cycle	No	No	No	Yes
Fire multiple instantiations	No	Yes	Yes	Yes
Exploit parallelism	No	No	Yes	Yes
Special case inhibition	Yes	No	Yes	No

Figure 2.1 Comparison of HSP with related PSAs

Production Memory

The three comparison PSAs all have a special action which in some way adds a new production to PM, making PM dynamic. HSP does not, for several reasons. The complexity of the implementation would be substantially increased, and the gap between HSP and HSII would be widened, making comparisons less meaningful.[4]

[4] The decision not to add productions dynamically is consistent with what is known about human information processing. Several seconds (the length of a speech utterance) is simply not long enough for appreciable storage into long-term memory [Newell and Simon, 1972].

Working Memory

In contrast to simple symbolic list structures (e.g., (a b c) or (a (b (c)) (d))) used by the other PSAs, HSP uses an attribute-value structure.[5] The format for a WM element field is identifier/value, with fields being positionally defined if identifier/ is omitted. Arbitrary fields (or subsets of fields) of a WM element can be easily referenced in a condition element or action element, and new fields can be created dynamically. This added flexibility allows efficient grouping of context into fewer WM elements than would otherwise be possible. For example, this means less searching is necessary during condition evaluation.[6]

The following is an example of an HSP WM element to represent the hypothesis that the word "BOOK" was spoken, beginning at .17 to .23 seconds into the utterance and ending at .58 to .62 seconds, with the validity of the hypothesis believed to be 60%.

```
< HYP WRD "BOOK" BTIME/(20 3) ETIME/(60 2) VLD/60 >
```

This example illustrates most of the field value types: HYP and WRD are symbolic field values; "BOOK" translates internally into an integer (the index of BOOK in the word lexicon), and the VLD (validity) is an integer; the begin and end time field values are lists of two integers (time and range).

The last value type, not shown in the above example, is an explicit reference to another WM element. Inter-element references need to be represented somehow, and doing so explicitly is more satisfactory than implicitly (e.g., by having the two elements share a unique subfield, or by having a special third element to represent the linkage). Also, as discussed in Chapter 4, these explicit references have a strong positive effect on time efficiency.

The attribute-value and explicit reference features were borrowed almost unconsciously from HSII where information in the Blackboard is structured as hyps and links (the analog of HSP WM elements), with some fields being pointers to other related hyps or links.

Conditions

All conditions in HSP begin with an element which directly tests the nature of a change to WM (i.e., which type of change, and the identifier of the changed field). Changes to WM are themselves explicitly symbolized in the same form as WM elements, although they are

---

[5] However, because of flexibility in matching, PSG is able to get most of the effect of an attribute-value structure with: ((a1 v1) (a2 v2) ...).

[6] This WM element structure is very similar to the Parameterized Structural Representations of Hayes-Roth [1973].

not stored in WM.[7] This condition on a change follows naturally from the HSII architecture in which KSs are required to declare (for purposes of efficient monitoring) which sorts of changes they wish to react to. In HSP, this initial condition helps efficiency in two ways: it allows production evaluation filters to be built (i.e., associations from characteristics of a change to relevant productions, analogous to the HSII change-monitoring mechanisms), and it eliminates some WM searching during production evaluation by pre-binding a variable to the changed WM element (which is presumably being matched by the production if it first matched the change itself).[8] But it is more than just an efficiency mechanism; it actually encodes some additional knowledge. In effect, it says "this is the only relevant change for this condition". This saves the inefficiencies of monitoring for and responding to irrelevant changes. But sometimes, when more than one sort of change is relevant, it forces wasteful duplication of the production. For example, a sequence recognition rule requires a separate production for each constituent, representing the case of that constituent being the last to appear. This duplication is significant but bearable in the KSs translated for HSP, but it is expected nevertheless to be a serious problem in a more complex system.

This feature of including a condition element on a change provides another quite important function; namely, it solves the excitatory instability problem. A production will fire only when some change has just occurred to WM which makes the production condition true. In successive cycles the production will not continue to fire even though the same WM state remains true, because the relevant change has not just occurred. PSNLST and OPS eliminate excitatory instability by not refiring the same instantiation of a production. PSG provides no solution, leaving it to the user to explicitly mark WM elements in every action to keep the production from refiring on the next cycle.

The HSP architecture permits special predicates to apply to integer-valued WM element fields. These are expressions surrounded by [ ] and built from the arithmetic relations > and < and operations +, -, *, /, MAX, MIN and ABS. Further, such predicates can be composed to form compound predicates with AND, OR and NOT. (This NOT is different from the one for negation of an entire condition element). For example, the predicate [NOT > $BT+15] tests a WM element integer field to see if its value is not greater than the value of variable $BT plus 15.

PSNLST and OPS go even further than HSP by allowing an arbitrary LISP function as a predicate. This may be inappropriate because it gives too much power to the condition evaluation, and if exploited to its fullest could subvert the basic PSA philosophy.

The attribute-value structure in HSP allows a condition to be specified on an arbitrary

[7] An early version of HSP did store change elements in WM, opening up some interesting possibilities for the PS monitoring its own behavior. However, the feature had to be abandoned for implementation reasons.

[8] PSNLST uses the same sort of association from changes to relevant productions and also does some pre-binding to variables in a production condition based on the change, but the change is not explicitly matched in the production condition.

subset of the fields of some WM element. For example, the condition element < HYP WRD VLD/[> 80] > matches all word hyps whose validity is greater than 80, regardless of what other fields they may have. This is essential, given that information tends to get added as extra fields of existing WM elements rather than as separate WM elements. PSG and OPS have a partial capability for matching subparts of WM elements, in that a condition can apply to an initial subsequence of subelements (i.e., the tail of an element can be ignored).

Although early versions of HSP followed the lead of the other PSAs in providing disjunctions within conditions, disjunctions were eliminated in an intermediate design iteration. This simplified the interpreter and increased parallelism (but at the cost of some increased redundancy, both in space and time). It eliminated the (minor) confusion of a second source of multiple firings of a single production (the first being multiple matching WM elements for some condition element). Finally, it made production counts as a measure of knowledge content more meaningful since it prevents two productions with substantially different conditions from masquerading as one just because their actions are the same.

Negated conjunctions were ruled out to simplify implementation, but also out of a vague feeling that they give more power to a single production than is appropriate.[9] Besides, a negated conjunction such as NOT (A B C) can be split ir separate productions with the following conjunctions: NOT A , A NOT B and A B NOT C. ith the conjunctions written in this way the three productions are mutually exclusive, avoiding multiple firing problems. Rychener [1976] cites negated conjunctions as an essential feature of PSNLST, being used .4 times per production on the average. Some of these uses can be expressed in HSP as a single negated condition element.[10] Yet there are other important uses of negated conjunctions in PSNLST that are not conveniently representable in HSP, requiring instead several coordinated productions acting over several PS cycles.

In HSP, if a condition element is preceeded by $V=, then one of two things may happen. If the variable $V is unbound in any previous (to the left) condition element, then a successful match of the current condition element will bind the matched WM element to $V. If on the other hand the variable is already bound, the condition element is applied exclusively to the WM element reference bound to $V (i.e., no WM searching occurs). Most often this case results from the variable having been bound to an explicit WM element reference contained in an earlier-matched WM element.

---

[9] Behind this is a concern that too powerful a condition language will defeat the basic PSA property of "immediate" recognition.

[10] For example, the HSP condition: <a 3> NOT <a [>3] > is true if the element <a 3> is in WM, but no similar element with a number greater than 3 is also. In PSNLST a negated conjunction is required to represent this since the ">" predicate is not built into the architecture.

Actions

There are seven action primitives in HSP:

NEW	creates a new element for WM
DEL	deletes an element from WM
MOD	replaces in a WM element the value of the field with a given identifier
MOD.ADD	adds to a specified list field of a WM element
MOD.ADDE	adds to the end of a specified list field
MOD.REP	makes a replacement within a specified list field
MOD.DEL	deletes from a specified list field

PSNLST and OPS have only a create and delete for whole WM elements. This is of course adequate, since any small change can be made by deleting the old element and creating a new one which is a copy of the old except for the change. But the large size of most elements in HSP makes this undesirable, both for efficiency reasons and because copying of an element is not possible when only a few subfields have been matched (as is often the case).

It should come as no surprise that the set of HSP action primitives closely mimics the set of Blackboard modification routines in HSII. Note that a MOD, MOD.ADD or MOD.ADDE operation on a non-existing field will create the field; i.e., this is how dynamic creation of new fields is effected.

Control

A feature of HSP that distinguishes it strongly from the others is the absence of any conflict resolution process. All productions which evaluate to True in a given cycle, no matter how many, are allowed to execute.[11] The motivation for this design choice comes largely from the example of HSII; the existence of multiple, independent sources of knowledge (productions in HSP) makes multiple production firings a natural choice.

Another reason for doing without conflict resolution is a desire to minimize the knowledge built into the PS interpreter and simply let the production conditions take care of themselves. The OPS designers disagree with the desirability of this. McDermott and Forgy [1978] express a concern that extending production conditions to resolve their own conflicts will severely restrict the system's ability to learn. Thus they opt for a thorough, carefully designed conflict resolution mechanism [Forgy and McDermott, 1977]. Learning in HSP is a moot point; but by analogy, absence of conflict resolution may well have a serious negative impact on the problem of augmentation.

The absence of a conflict resolution mechanism does cause problems for HSP. But they seem to be surmountable, often requiring a few extra coordinated productions operating

---

[11] Although a very large number might cause some concern because of a loss of control over directionality. I.e., the system's ability to rapidly shift its focus of attention would be hampered.

over several PS cycles. These extra productions usually make use of an explicit delay of some small number of PS cycles. This n-cycle delay is implemented with a chain of productions that simply mark time, waiting until some related activity is known to be finished.

HSP is alone in allowing multiple productions to fire in a single cycle. The other three architectures use their conflict resolution to select a unique production to fire, resorting to arbitrary choice when necessary. Although multiple productions may fire in a single cycle, there is still a global synchronization at the end of every cycle. All productions must have been evaluated before any action may be executed. (During evaluation, actions are "interpreted" to obtain symbolizations of the changes indicated in the actions, then when all evaluation is completed the changes are actually made). This has direct implications for parallelism and synchronization of parallel activity, as will be seen in Chapter 5.

Firing multiple instantiations of the same production means that the action of a single production may be executed multiple times -- once for each possible set of bindings to variables in the condition that make it True. HSII KSs exhibit an exact analog of this property; in fact, its use seems almost mandatory for the case of speech knowledge operating at multiple locations in the utterance-time dimension. For example, a production which responds to a new word "BOOK" should fire twice if two "BOOK"s appear at different utterance times.[12] Of the comparison PSAs, PSG is the only one which doesn't have this property of iterating through all instantiations of a production; current philosophy seems to regard it as an essential convenience.

A feature which separates HSP from the others is the use of parallelism in the underlying system to speed up evaluation.[13] No more will be said here about parallelism since Chapter 5 is devoted to it.

HSP has no mechanism for special case inhibition; i.e., a mechanism that prevents a production from firing even though true when a more special (however that is defined)[14] production is also true. (The other PSAs obtain special case inhibition through their conflict resolution). Doing without special case inhibition eliminates some mechanism from the PS interpreter, but this is a relatively minor advantage; the main reason for the decision was to obtain as high a degree of parallelism in the evaluation as possible. Any

[12] The need to focus activity of the system provides a basis for favoring certain time regions, but focussing in HSP should be controlled by augmenting production conditions to be sensitive to the time region, not by having a separate mechanism.

[13] The original version of OPS was designed to exploit parallelism, although its implementation was not on a parallel machine. The most recent OPS version has given up its inclination toward parallelism in order to be more efficient on a uniprocessor.

[14] E.g.: Pa is a special case of Pb if it has all the condition elements of Pb plus some extra ones.

scheme for doing the general-special case checking would probably result in a significant reduction in parallelism.

Giving up special case inhibition has some serious consequences, both for programming convenience and for the learning of new productions without modification of existing productions (though this is a moot issue for HSP with its static PM). However, there are ways for HSP to get around the programming inconvenience: general case productions can be augmented to make them special, i.e. to make them fire only when no special case production is true; or both general and special productions can be allowed to fire, if there are other productions which can detect this and favor the special case result.

Chapter 3

A Comparison of Representation in HSII and HSP

This chapter compares several aspects of representation of knowledge in HSII and HSP. It is based on 12 KSs which were translated from their HSII form to HSP productions, with varying degrees of completeness. Figure 3.1 shows these KSs, all of which existed in the January 1976 version of HSII.[1] The following four KSs, also part of that HSII version, were not translated to HSP: PSYN and CSEG for mapping segments into phones;[2] FOCUS for directionality control; and the postdiction KS of the SASS module.

SASS module (Syntax and Semantics)	
RECOG	Recognition of phrases from words and subphrases
RESPELL	Spelling of phrases into words and subphrases
PREDICT	Prediction of adjacent words and phrases
SASS module (newer version)	
RECOG	Recognition of phrases from words and subphrases
POMOW module	
MOW	Recognition of words from syllables
POM	Recognition of syllables from segments
WOMOS module	
WOM	Spelling of words into syllables
MOS	Spelling of syllables into phonemes
POSSE module	Phone - surface-phoneme synchronization
TIME	
SEARCH	
SEG	For inputting segments into Blackboard
RPOL	Rating propagation policy

Figure 3.1 The HSII KSs that were translated to HSP

The POM KS was translated with particular attention to completeness, and thus provides a focal point for most of the detailed comparisons that follow. POM's function is to recognize likely syllables from the speech segments at the next lower level of the Blackboard. It is a complex, multi-phase process, using three intermediate levels of

[1] This is configuration C1 in [Lesser and Erman, 1977]. For more information about these KSs consult the following: SASS [Mostow and Hayes-Roth, 1978; Hayes-Roth, Mostow and Fox, 1977]; MOW and POM [Smith, 1976]; WOM, MOS and POSSE [Cronk and Erman, 1976]; RPOL [Hayes-Roth, Erman and Lesser, 1976].

[2] These two were eliminated from HSII soon after January 1976 by changing POM to work directly from segments rather than phones.

representation between the segments and syllables. It contains a wide variety of knowledge, possibly more so than any other KS, and thus exercises a good deal of the HSP architecture.

In spite of the attempt to translate POM with absolute fidelity, some deviations were unavoidable. Most of them can be justified by the architectural differences of HSII and HSP; the remaining few are unimportant. What is needed is some assurance that no significant differences do exist; i.e., that the translation to HSP was adequate. The first evidence for this lies in the nature of the translation process itself. During translation the HSII KS code was studied statement by statement, and thus it is unlikely that any major omissions or mistranslations occurred. The second bit of evidence is more direct: in the single test run[3] the HSII and HSP systems produced the same five alternative syllable hypotheses as output.[4] The syllable ratings do not match exactly since different methods are used for calculating the final rating. (This is one of the aforementioned unimportant differences). However, the rank orderings are reasonably consistent.

This chapter proceeds now to comparisons of several representational aspects that distinguish HSII and HSP. These aspects were chosen because they relate (directly or indirectly) to one of the implementation problems introduced in Chapter 1, or perhaps because they present a particularly striking difference between HSII and HSP. To aid in coherence, these comparisons are divided into three categories: long-term memory (containing the bulk of the material), working memory, and control.

3.1 Long-Term Memory

By long-term we mean having a scope spanning more than the understanding of a single utterance. In the case of a speech understanding system remembering the context of an entire discourse, there might be an issue about whether to call this long-term or short-term knowledge. However, HSII and HSP deal only with isolated utterances. Thus the long-term knowledge is the static, general speech knowledge. The short-term memory is empty to begin an utterance, and is dynamically built up during the processing of the utterance.

There are quite a number of comparative points to be made about long-term knowledge representation, and each will now be considered in turn.

[3] The nature of the test run is explained in the following chapter.

[4] Actually, POM outputs syllable classes, called _syltypes_, but this is not important in this context.

3.1.1 Adequacy

The basic evidence for adequacy is that a large number (12) of HSII KSs were translated to HSP productions. However, there are nearly 20 other KSs that have been used at one time or another in HSII [Lesser and Erman, 1977]. Most of these were simply not yet in existence at the time the HSP translations were done. Based on a moderate understanding of how these other KSs work, there is no reason to believe they would be more difficult to represent in HSP than the 12 that were translated.[5] Efficiency is another matter, as discussed in Chapters 3 and 4.

This is not to say there were no difficulties in the translation, or that the resulting productions do not have some awkward aspects. For example:

> An HSP production cannot iterate over WM element list fields of arbitrary length. A separate production is required for each possible list length, while in HSII a single iteration statement suffices. This apparent failure of the PSA is an artifact of the mimicking in HSP of HSII's explicit references between hyps. If references in HSP were individually represented as WM elements, iteration would be handled naturally by the multiple firing of a single production.
>
> Controlling duplicate actions is an occasional difficulty. When the changes in question are simple modifications (i.e., the MOD action primitive), the system automatically weeds out the duplicates so that multiple production firings do not occur on the following cycle. When the changes are NEW, DEL, MOD, ADD, etc., the system provides no help. The real problem with NEW elements is not detecting a duplicate, but the lack of any basis for favoring one of the duplicates (while the others are deleted). It might be feasible to automatically check for duplicates even in this case, but it would be expensive. For example, every NEW WM element would trigger a match against all of WM. An alternative solution is used by the HSP POM KS to eliminate duplicate syllable recognition. An extra cycle is taken which effectively represents in WM the fact that a new syllable is to be created, but distinguishes the two possible sources. Then the following cycle is able to detect duplication, and simply always favors one source over the other.
>
> There is currently no way in HSP to keep a tally (in WM) of some event being monitored. For example, a production counting new word hyps would add one to a count in WM every time a new word appeared. But if several appeared on the same cycle, the count would still be only increased by one (although redundantly). This tallying capability is representative of a class of operations which are crucial

[5] There is some direct positive evidence for one of the more suspect untranslated KSs, namely the word verifier which uses the WIZARD procedure [McKeown, 1977]. HSP POM contains productions for searching a syllable state transition network which is essentially the same as the word networks used by WIZARD. The POM productions would not carry over exactly because the WIZARD networks have longer paths with less favorable combinatorics. But the POM productions do show basic adequacy for WIZARD.

for implementing directionality in HSP (see Section 3.3.2). Perhaps the right answer is the simple expedient of adding an "increment" change primitive.

Arithmetic expressions in HSP often exhibit substantial redundancy within a single production. Some means of saving results of common subexpressions (e.g., by assignment to a local variable) would be helpful.

Taken together, these difficulties, and the few others not described here, are not serious enough to refute the claim of adequacy, though they do take their toll in other problem areas such as efficiency. Furthermore, there seems to be a good possibility that iteration on the design of the HSP architecture could remove most of these difficulties without deviating from basic PSA philosophy.

3.1.2 Simplicity of the Language

While there are conventions for HSII KSs, including the provision in the HSII kernel of many primitives for accessing the Blackboard, KSs tend to use most of the expressive power that SAIL provides: character-string manipulation, sets, lists, associative structures and conditional assembly. The language provided by HSP for writing productions is a good deal simpler than SAIL. Figure 3.2 shows a comparison of four aspects of the language environment for KSs: the number of primitives (operators, pre-defined functions, constants), the number of data types, the size of the runtime support, and the size of the "compiler" for the language.

	HSII	HSP	HSII/HSP
Primitives	310	53	6
underlying system	130		
kernel	180		
Data types	10	5	2
Runtime support (Kbits)	2212	494	4
underlying system	457	203	2
kernel	1755	291	6
Compiler (Kbits)	3100	264	15

Figure 3.2 Comparison of KS languages in HSII and HSP

For HSII the primitive count is split into basic SAIL plus interface to the HSII kernel; no such split makes sense for HSP since productions as a rule do not contain any basic L* code. The SAIL primitive counts do not include I/O formatting or conditional compilation functions, nor any primitives that a KS would not reasonably be expected to use. The runtime support for HSP includes the production interpreter (40 Kbits); SAIL has no such interpreter. As a result its compiler's task is more difficult than HSP's. HSII runtime support excludes 817 Kbits for debugging facilities in the kernel. The HSP "runtime support" and "compiler" in HSP are just pieces of a single integrated system built up from a basic L* system of 203 Kbits, but the size of each above excludes storage of external names (141 Kbits) and utilities for editing, debugging, etc. (49 Kbits).

In summary, the HSP production language can be said to be roughly 5 times simpler than HSII's KS language, with the exact factor depending on how it is measured. This simplicity is a positive feature in so far as it helps with problems such as debugging and augmentation. Yet is raises some concerns about adequacy, and even stronger ones about efficiency.

3.1.3 Declarative Knowledge and Multiple Use

The distinction between declarative and procedural encodings of knowledge has been the basis for a long-standing dispute in the field of artificial intelligence. Winograd [1975, p.186] provides a good overview of these two opposing viewpoints:

> *The proceduralists assert that our knowledge is primarily a "knowing how". ... What a person (or robot) knows ... is coexistent with this set of programs for operating with it. ... The declarativists, on the other hand, do not believe that knowledge of a subject is intimately bound with the procedures for its use. They see intelligence as resting on two bases: a quite general set of procedures for manipulating facts of all sorts, and a set of specific facts describing particular knowledge domains.*

Winograd goes on to describe some of the advantages on each side, and concludes that the declarative and procedural formalisms are endpoints on a spectrum of modularity vs. interaction. The ultimate in modularity is exemplified by a set of logically independent mathematical axioms, a pure declarative representation. At the opposite end, programming deals explicitly with interactions. The ultimate answer may lie in an appropriate synthesis of the two extremes. Indeed, a PSA can be seen as an attempt to recover some benefits of modularity in a pure procedural encoding by forcing all interactions through a single working memory.

The HSII KSs use a combination of procedural and declarative representation for long-term knowledge. The procedural is in the form of SAIL code; the declarative consists of variables and arrays whose contents are initialized from files at the beginning of a run. In the HSII POM KS, for example, there are arrays for such things as segment vowel probabilities, legal syllable state transitions, state transition probabilities and spellings of syllables as state sequences. Declarative structures such as these account for 30% of the total space in POM, but their dynamic use is greater than that figure implies.[6]

[6] Unfortunately, no dynamic data on use of knowledge in HSII could be obtained.

HSP, on the other hand, has no declarative long-term memory -- all long-term knowledge must be encoded procedurally as productions.[7] This leads to a number of difficulties for HSP. The speech task in general seems to require a large number of knowledge bases that are representable conveniently in declarative form; e.g., tables of probabilities, similarity matrices, dictionaries of spellings, and grammars. When these knowledge bases must be represented procedurally, there are serious problems with efficiency and multiple use of the knowledge.[8]

Virtually all of the instances of large uniform data bases can be represented efficiently in HSII by packing into arrays; and cost considerably more when represented as HSP productions. The following are examples of this, with comparative space costs given:

> Vowel probabilities ("A"-type, "I"-type, and "U"-type) for each of the 43 possible segments.
>
> > HSP uses 43 productions, each of which modifies a new segment hyp to have three new fields with the "A", "I", and "U" vowel probabilities. Each production requires 10 words;[9], for a total of 7 Kbits. , HSII packs the probabilities into a single 43 entry array, for a total of 1.8 Kbits.
>
> State transition probabilities: the probability that state _y_ follows state _x_ given that segment _z_ is one of the alternatives adjacent in utterance time to state _x_.
>
> > If represented in a standard array, a total of 6 x 6 x 43 = 1548 entries would be required for the left syllable states, and an equal number for the right. If a sparse encoding were used, the number could be somewhat smaller since not all pairs of states represent legal transitions, plus many zero entries could be omitted. In fact, HSII POM stores only 883 probabilities, each packed into 18 bits, for a total of 16 Kbits. In HSP, each probability value is encoded by a production which responds to a special WM element containing the context of states and segment; if the context is the one that the production encodes, the probability will be stored into a field of the new state hyp. The high cost of 10 words (160 bits) per production forced the use of a cutoff on the probability value: only values of .4 or above are represented. This brings the number of productions

[7] Winograd's natural language understanding system [Winograd, 1972] is a prime example of procedural encoding. But it also contains some declarative long-term knowledge that is used quite heavily, e.g., a set of syntactic features associated with each dictionary word, and a semantic feature network used for an initial phase of semantic analysis. Thus HSP is more procedural than Winograd's system.

[8] A side problem was the large number of productions which had to be defined. This was solved by making modifications to the auxiliary programs that HSII uses to initialize the declarative structures, making them output productions in HSP format instead. This was done for POM, MOW, WOMOS and SASS.

[9] A condition procedure is used to save space -- see the end of Section 3.1.4.

down to a manageable (though still excessive) 589, for a total of 94 Kbits. The impact on performance of omitting all values less than .4 should be minor, but is unknown.

The second major difficulty with procedurally encoded knowledge is allowing for multiple use. When knowledge is encoded in a production it is tied to some particular condition, and thus cannot be applied under different conditions without duplicating it in other productions. Declarative encodings do not have this problem since they are divorced from control information, and any procedural unit that has access to the data can make use of it.

There are several methods that HSP uses for dealing with the problem of multiple use:

(1) Duplicate the knowledge. When the number of uses is small, it does little harm to just duplicate the knowledge in the several productions that use it.[10]

An example of this in HSP is the dictionary of word spellings. The MOW KS uses these for recognizing words from syllables, while the WOM KS uses them in the opposite direction for prediction of syllables from words. Thus the spellings of words as syllables are duplicated in HSP, whereas HSII has a single copy in declarative form globally accessible by both KSs.

Often when knowledge must be duplicated in a number of productions, it is possible to use condition and action procedures (see the end of Section 3.1.4) to maintain a single shared copy. Of course, this is only possible if the different uses are just minor variations, but they often are. And this sharing does solve the multiple use problem without deviating from pure procedural encoding.

(2) Subroutines. Knowledge with multiple uses can be encoded in productions that respond to a special WM condition (the subroutine call) and make the knowledge available (to the caller) by appropriate WM changes.

This method is used for the syllable state transition probabilities in POM. There are three conditions for use of the knowledge, depending on whether the segment in question is the first, second, or third alternative for a given time region. These three productions each create a special WM element which contains the context for determining the probability. The appropriate probability production responds to this new WM element and, using the information recorded there, returns the probability as a change to a known field of the new state hyp.

(3) Copying into WM. In the case where multiple conditions for use of some knowledge share a common subcondition, that common condition can trigger a production which deposits the knowledge into WM. It is then available in declarative form for any use.

[10] The problem of updating the multiple copies could be serious in some circumstances, although it was not in the KSs translated.

A version of SASS was written to try this approach for the rules of the grammar. The appearance in WM of a word or phrase triggers the copying into WM of all rules that have it as a constituent (one copying production per rule). These WM elements representing grammar rules remain for only a single cycle,[11] which is long enough for use by any of the recognition, respelling, prediction, or postdiction productions that care to respond to the original word or phrase. Time constraints did not permit debugging and running of this SASS version, so it is not known whether the overhead of the continual creation and deletion of grammar rules in WM is prohibitive.

These means for controlling the problems of multiple use are sufficient for the current HSP system, but it remains to be seen whether or not they break down when more complex systems are attempted.

It seems appropriate at least to entertain the possibility of adding a facility for long-term declarative knowledge to the HSP architecture. Such a facility was spurned during development of HSP to keep the architectural comparisons pure. But if HSP were a development system rather than a research tool, immediate efficiency concerns would probably force a number of compromises such as this.

The syllable state transition probabilities in POM provide an example. As mentioned above, the current scheme uses two PS cycles: the first involves 6 productions (3 for the alternative segment positions with left syllable states, and 3 similarly for right) which create a special WM element containing the relevant context. This creation triggers on the next cycle some subset of the 589 productions that encode the probability values, at most one of which will actually fire.

Suppose that the simple expression language used in HSP productions were augmented with a facility for accessing a multidimensional array. Then there could be just a single cycle: 6 productions which bind the relevant context to variables and then access the probability from a three-dimensional array. This would save the space for the 589 separate productions (about 100 Kbits), and remove the extra cycle and the creation and deletion of the special WM elements. It would also permit inclusion of the array values below .4 which the current scheme eliminates. Storing the full array would take about 50 Kbits in HSP, but this could be reduced substantially by packing and using a sparse encoding.

3.1.4 Space Efficiency

A comparison of space requirements for representation of long-term knowledge in HSII and HSP is not an easy task. The two systems have many differences incidental to the architectural differences we are interested in comparing. Thus we cannot be sure that the

[11] I.e., there is a production which responds to the appearance of such an element in WM and deletes it on the following cycle.

sizes don't reflect programmer differences, degree of optimization of the language, or improvements due to HSP being an extra iteration beyond HSII (to name just a few). Nevertheless, the data do provide a comparison that is accurate enough for our purposes.

Although 12 HSII KSs were translated to HSP, usable space comparisons could be obtained for only two: POM and RPOL, as shown below in Figure 3.3. Most of the other 10 are incomparable because of simplifications introduced during translation. Several actually ceased to exist as HSII KSs, making it too difficult to obtain good HSII size data.

	size (Kbits)		ratio
KS	HSII	HSP	HSP/HSII
POM	220	238	1.1
RPOL	30	30	1.0

Figure 3.3 Space comparison of long-term knowledge in HSII and HSP KSs

Because of the special problems declarative knowledge poses for HSP, space accounting needs to be done separately for procedural and declarative HSII knowledge. In the figure above, the ratios are very close to 1; but RPOL is virtually all procedural knowledge, and POM is mostly so (its declarative/procedural split is .3/.7). As we shall see below, declarative knowledge alone yields HSP/HSII space ratios significantly larger than 1.

A detailed space analysis was carried out on the POM KS, taking care to separate declarative from procedural knowledge. This is shown in Figure 3.4, with sizes broken down into a number of functional categories. The categories are ones the HSP productions naturally form. HSII POM had to be mapped into these categories on a statement-by-statement basis, though the mapping was fairly clean. The procedural sizes include modest amounts of local working memory (e.g., local variables) since this could not be easily separated from the long-term knowledge.[12] Although all long-term HSP knowledge is procedural, only that part translated from HSII procedural knowledge appears in this comparison.

[12] This would be inappropriate for KSs such as SASS which have large amounts of local working memory.

	size (Kbits)		ratio
functional unit	HSII	HSP	HSP/HSII
Utterance boundary handling	7.3	5.7	.8
Gap (silence) handling	12.1	5.3	.4
Segment vowel probabilities	0	1.0	--
Total vowel probabilities	2.4	2.7	1.1
Syllable nucleus finding	24.2	9.4	.4
Nucleus context building	22.4	14.9	.7
Segment rating normalization	1.7	3.8	2.2
Combined vowel probabilities	2.8	5.8	2.1
Syllable state transitions	64.7	31.6	.5
Endstate boundary time setting	2.9	5.1	1.8
Syllable recognition	16.6	20.7	1.2
Total	157	106	.7
Initialization	85.3	0	0
Display	27.9	0	0

Figure 3.4 Space comparison of procedural long-term knowledge in HSII and HSP POM

Note that HSP has nothing under the Initialization category. The function of initialization in HSII is to set up the contents of some variables and arrays with long-term knowledge (from auxiliary files), and to clear the contents of others being used for KS-local working memory. HSP has no need for either of these functions. The Display category is also given separately since display productions were left out of HSP POM to reduce the total translation effort. Also, the display facilities are less essential in HSP than HSII due to the central tracing facility in HSP. The internal workings of an HSII KS are not so easy to trace since they are not under interpretive control, although it is possible to trace calls to HSII kernel functions (e.g., Blackboard accesses).

The "segment rating normalization" and "combined vowel probabilities", though pure procedural, have unusually high HSP/HSII ratios. This is due to the inability of an HSP production to deal with WM element list fields of arbitrary length. In both the above cases some operation is being performed on all the alternative segment hyps at a given time position, as listed in a field of an "option segment" hyp. This requires a separate production for each possible total number of alternatives. In this case the maximum number is only three, by convention with the KS which creates the segment hyps. If it were larger, as it may well be in other cases, the space costs would be much more serious. By comparison, a single iteration statement suffices in HSII to handle any number of alternatives.

There are a couple of possible explanations for the 30% decrease in total size from HSII to HSP. First, HSP productions can be more compactly encoded because they are interpreted. Secondly, HSP can represent simple conditions and complex searches of global working memory more concisely than HSII. But the substantial variation in the space ratio from .4 to 2.2 across the separate categories belies such simplistic explanations, and a complete explanation is buried in many separate details.

Data on long-term declarative knowledge, shown in Figure 3.5, is available from all of the translated KSs since it can easily be estimated from source listings in both systems. The results are given individually for all the large structures, and each is classified according to its type. Recall that the HSP data actually refers to that part of its all-procedural long-term knowledge which is a translation of declarative HSII knowledge.

	source	type	size (Kbits) HSII	HSP	ratio HSP/HSII
(1)	POSSE similarity matrix	array	20	25	1.3
(2)	POM vowel prob	array	1.8	6.9	3.8
(3)	WOSEQ word adjacency	bit array	1000	1100	1.1
(4)	MOW syl-word dict	spellings	30	30	1.0
(5)	POM state-syl dict	spellings	29.6	24.3	.8
(6)	SASS grammar	network	30	35	1.2
(7)	POM syl state trans	network	31.1	100	3.2
	Total (excluding (3))		140	220	1.6

Figure 3.5 Space comparison of declarative long-term knowledge in HSII and HSP

The WOSEQ bit array (3) is an unusual case deserving special explanation. WOSEQ, the word sequence hypothesizer, uses an n x n bit matrix (where n = the number of words in the vocabulary, or 1000 in a full HSII system) to provide a fast test for grammatical adjacency of words. Since each entry requires a production of about 64 bits in HSP, the consequences could be disastrous. However, the sparseness of the matrix saves the day: only 17,000 productions (rather than a million) are necessary since the matrix has ones in only 1.7% of its entries.

Excluding (3), the total ratio comes to 1.6; i.e., a 60% size increase in translating from HSII to HSP. But there is considerable variation among the individual cases. The differences between the ratios for (1) and (2) can be completely explained by the fact that array (2) is totally represented in HSP, while array (1) is not since it has only 35% non-zero entries. HSP's representation of arrays acts as a sparse encoding mechanism: productions are required only for the non-zero entries while the zero values are handled as the default.[13] The relatively small ratios for (4) and (5) (spellings, or arrays of strings) can be explained by a peculiarity of SAIL: there is a large (72 bit) fixed overhead for string representation.

The HSP sizes for both knowledge types (Figures 3.4 and 3.5) are biased upward due to non-optimal encoding of productions. In particular, HSP condition elements are represented using the same type of list structure as WM elements, requiring 3 words per field. A careful encoding could reduce this cost by a factor of between 2 and 3, giving an overall reduction in space of as much as a factor of 2. However, it should also be noted that the SAIL compiler does not produce highly optimal code, so that a similar (though probably smaller) bias exists in the HSII data.

Other measures besides bit counts help to fill out the picture for procedural knowledge. One is token counts; i.e., counts of lexical atoms, as would be recognized by the KS language compiler. They presumably provide an indication of program size as perceived by a human. The other is statement counts. For SAIL, the usual definition of statement is used; for HSP, the sum of production, condition element, and action element definitions is used, corresponding to a view of condition and action elements as simple statements and productions as compound statements. These counts (shown below in Figure 3.6) mainly provide additional support for the bit counts of Figure 3.4 -- the correlation is good for the most part. Functional units containing declarative knowledge have been omitted since token and statement counts do not make sense for them.

[13] Since HSP array representations make use of the PM index efficiency mechanism (described in the following chapter), it might be appropriate to include the PM index space even though it is inessential (i.e., is only for time efficiency). In general, a one-dimensional array requires about 100 bits per production for the PM index. For example, the ratio for (2) would increase from 3.8 to 7 if we included PM index costs. However, we chose not to do so throughout Figure 3.5.

	token count		ratio	statement count		ratio
functional unit	HSII	HSP	HSP/HSII	HSII	HSP(P+CE+AE)	HSP/HSII
Utterance boundary handling	291	267	.9	33	2+ 6+ 10	.6
Gap (silence) handling	419	244	.6	50	3+ 9+ 4	.3
Total vowel probabilities	119	146	1.2	14	3+ 5+ 3	.8
Syllable nucleus finding	551	387	.7	66	8+ 23+ 11	.6
Nucleus context building	964	817	.8	121	22+ 39+ 6	.6
Segment rating normalization	60	203	3.4	5	3+ 8+ 6	3.4
Combined vowel probabilities	95	312	3.3	8	3+ 8+ 9	2.5
Endstate boundary time setting	123	239	1.9	13	4+ 12+ 4	1.5
Total	2622	2615	1.0	310	48+110+ 53	.7
Initialization	2627	0	0	337	0+ 0+ 0	0
Display	1367	0	0	163	0+ 0+ 0	0

Figure 3.6 Token and statement comparison of procedural knowledge in HSII and HSP POM

Space costs in HSP would have been much higher if it weren't for the capability of sharing common subparts among productions. Any sequence of condition elements (even a single element) can be defined in a named list (called a condition procedure), and then this single copy can be referenced in multiple productions.[14] These procedures can even be parameterized for cases where multiple uses are similar but not identical. Action procedures provide the analogous function for action elements. These capabilities are used heavily: the 906 productions of the POM KS have a total of 2214 condition elements and 1457 action elements, but only 10% of condition and 8% of action elements are separately represented. Thus a factor of about 10 in space is saved in this manner.

3.1.5 Knowledge Unit Size

The HSII and HSP architectures display a large difference in the size of knowledge units that are evokable by global working memory conditions. For HSII we count an entire KS and its PRE (precondition) as a single unit,[15] while for HSP every individual production counts as a unit.

[14] The levels of substructure so produced are transparent to the production interpreter. The only effect on operation is a negligible slowdown due to traversing the extra levels, and to assigning of parameters to procedure variables.

[15] An HSII KS module combines one or more KSs (and associated PREs) into a larger unit, with some data structures shared among KSs. E.g., the POM and MOW KSs are actually combined in a single module called POMOW. Still, we will consider individual KSs as units because they are the smallest independently evokable units.

Figure 3.7 shows the number of HSP productions obtained in the translation of the 12 HSII KSs. The most reliable count is for the POM KS since POM was essentially completely translated. Simplifications were made in the translation of the other KSs, so their production counts are low -- perhaps as much as a factor of two in some cases. One reason there are so many productions is that many are simply an enumeration of cases for a single situation. For example, the WOM KS has a production for every word in the vocabulary, and POM has a production for every pair of right and left syllable halves that combine to form a valid syllable. Thus Figure 3.7 also shows production counts adjusted by discounting all such large enumerations.[16]

KS	count	adjusted count
SEG	50	4
POM	906	82
POSSE (TIME and SEARCH)	215	22
WOMOS (WOM and MOS)	254	30
MOW	491	15
SASS (early version)		
RECOG	305	90
RESPELL	234	76
PREDICT	192	78
General	30	30
SASS (new version)		
RECOG	172	17
General	30	30
RPOL	39	39
Time propagation	32	32
General	22	22
Total (early Sass)	2770	520
Total (new Sass)	2211	293

General includes: hyp merging, updating necessitated by new or deleted links, redundant link elimination, hyp time consistency checking.

Figure 3.7 Production counts for KSs in HSP

HSP knowledge units average about 250 times smaller than HSII units, or 30 to 50

[16] These adjusted counts are roughly the actual counts that would exist if some special facility for declarative long-term memory existed in HSP.

times smaller if the large enumerations are discounted.[17] These smaller units go hand-in-hand with a greater degree of data-directed control, as discussed in Section 3.3.1 below, permitting higher parallelism. It is speculated that smallness of units also has benefits for other problems such as performance analysis, but this thesis produces no evidence of this. HSII units are as large as they are partly because of a concern that increased interprocess communication costs with smaller units may swamp the increased parallelism [Fennell, 1975, pp 27-28]. But more significantly, in the current uniprocess HSII system there seems to be nothing to gain by making units smaller than necessary to obtain the desired granularity of directionality control (focussing).

3.1.6 Mixture of Condition and Action

The code for a HSII KS is typically a complex mixture of condition evaluation and action. Blackboard writing can occur at any time during KS execution and tends to be freely intermixed with Blackboard reading, and with reading and writing of local data structures. Even if with some KSs the writing did all happen at a particular place in their execution (e.g., the end), there is no feature of the language that would accentuate this, let alone enforce it. The opposite is true of HSP: every unit of knowledge (production) is strictly partitioned into pure condition (read-only) followed by pure action (writing). This difference between the two systems has much to do with exploiting parallelism (see Chapter 5).

3.2 Working Memory

The short-term or working memory holds the dynamic context of processing for a single utterance, and is cleared to begin a new utterance. Both HSII and HSP make a distinction between global and local working memory: the global is accessible to all (long-term) knowledge units and is the source of the data-directed control; the local is that used within a single knowledge unit and not accessible by other units. The two systems have a strong distinction in the ratio of global to local working memory use.

3.2.1 Local

HSII KSs have arbitrary, and often quite large, local data contexts made up of integer and real variables, arrays, sets, strings and associative structures. These local contexts

[17] The number of HSP productions for some of the KSs is strongly dependent on the vocabulary and grammar size, and the counts given above are based on a greatly simplified grammar and 100-word vocabulary selected from HSII's large grammar and 1000-word vocabulary. However, the adjusted counts are not affected, so the factor of 30 to 50 remains valid.

cannot be accessed by other KSs,[18] and thus are strongly distinguished from the Blackboard, which can be accessed by all KSs and is used for all KS communication (at least theoretically).[19] The HSP architecture is remarkable for an almost total absence of any such local data contexts. The only exceptions are a small variable memory which holds bindings of variables over the scope of a single production execution, and an even smaller temporary arithmetic expression memory. Thus virtually all dynamic data in HSP must be stored in the global, shared WM.

Some examples of how local data structures are used by HSII POM are given below. They show how HSII obtains gains in efficiency that are not possible in HSP.

> In POM, part of the condition for identifying a possible vowel segment as a syllable nucleus is the existence of a MXN (amplitude maximum) hyp within the time range of the segment. Testing this condition requires a Blackboard retrieval operation, and must be done more than once for some segments. Thus, for efficiency the condition is pre-computed for each segment and stored in a local boolean array.
>
> In POM, the complex procedure that identifies syllables operates in the context of a syllable nucleus segment and segment alternatives to its right and left (in utterance time). Before processing begins, the entire context is read into local arrays (the identity of the segments, their ratings, etc.) to obtain big savings on access costs.
>
> In the most recent SASS KS, the partial parse trees obtained in attempting to parse a language fragment are stored locally in a specially designed data structure. Earlier SASS versions used hyp and link structures on the Blackboard, and were hopelessly slow as a result.

The following chapter on time efficiency estimates the efficiency loss HSP suffers by having no local WM.

3.2.2 Global

Since the total burden for working memory in HSP is to be supported by the globally shared WM, there is a requirement for generality and flexibility in the WM element structure which is missing in the case of the HSII Blackboard.

In HSII, Blackboard elements are of only two types: hyps and links; and each has a

[18] This is not strictly true in the real HSII. Several KSs can be compiled into a single <u>module</u> and share context that is global to the module. In this case, the local context still cannot be accessed by KSs outside the module.

[19] Again, the real HSII has exceptions in that KSs occasionally share global data structures not in the Blackboard.

certain minimal size due to predefined fields. New fields may be added dynamically, but the predefined fields are immutable.

By comparison, elements in the HSP WM have no predefined fields[20] and thus may be used to conveniently represent arbitrary symbolic structures. Some of the elements do represent hyps and links, and these by convention have many of the same fields as their analogs in the HSII Blackboard. The remainder of the WM elements (about half of those created during the HSP test run) encode data and control information local to some KS. In addition to these whole WM elements, KSs also commonly attach extra fields to elements representing hyps and links.

Space efficiency is a potential sore point with WM in the current HSP system, since WM was designed for ease of implementation and flexibility rather than efficiency. In HSII the predefined fields of hyps and links are tightly packed into fixed fields, with some fields as small as a single bit. In HSP there are no predefined fields, so a field identifier must be stored with every field value. The values are not packed, so each takes a full word (16 bits). And finally, the basic WM element is a linked list structure, adding an extra link word to each field. The net result of these inefficiencies is a space ratio (in bits) of roughly 3 to 1 for HSP vs. HSII elements. Some of this difference is inessential implementation difference. But the HSP requirement for generality and flexibility does make it difficult to have predefined or packed fields.

Furthermore, the lack of local working memory in HSP results in more elements in the HSP WM than the HSII Blackboard in comparable runs. In the POM runs this ratio was about 2 to 1, but in other KSs which make heavier use of HSII local memory (e.g., the new SASS, or new MOW using WIZARD), it could be much larger -- a full order of magnitude or more. For example, the new SASS can in the course of a run easily create 1000 internal nodes representing partial parses. If this SASS version were implemented in HSP, those 1000 nodes would have to be represented somehow as WM elements.

3.3 Control

The final aspect of representation to be compared in this chapter is that of control. As with the previous aspects, there are some striking differences in the way HSII and HSP represent control. We distinguish low-level (intra-KS) control from high-level (directionality) since HSII has separate mechanisms for the two forms.

[20] Except that of CREATOR (a reference to the production that created it), but that is for diagnostic use only.

3.3.1 Low-level (Intra-KS) Control

A primary difference between HSII and HSP is the degree to which data-directed control is used. As shown previously in Figure 3.7, HSP has a great deal more data-directed knowledge units than a corresponding HSII system. For the POM + RPOL configuration the ratio is 500 to 1. And dynamic behavior presents a similar picture: in equivalent test runs HSII had only 4 invocations (1 of POM, 3 of RPOL), while HSP had 1944 production invocations, giving a ratio again of almost 500 to 1.

With so few invocations in the HSII run, it is apparent that virtually all the control is supplied by SAIL control structures within the KSs. For every data-directed invocation there were on average 250,000 machine instructions executed within the KS, corresponding to roughly 25,000 SAIL statements.

In the HSP run, each data-directed invocation resulted on average in 2.8 condition element evaluations and .3 action element executions. The condition evaluation process represents a form of backtracking search control;[21] action execution is unconditional and sequential. We can thus conclude that the control of condition evaluation dominates in HSP. Yet data-directed control has a very strong effect since it intervenes once for every 3 condition or action elements.

This high degree of data-directedness in HSP is principally responsible for high parallelism (see Chapter 5), and is speculated to also have a positive effect on other areas such as augmentation and performance analysis. However, it has a strong negative effect on time efficiency due to the high overhead of data-directed invocation (see Chapter 4).

3.3.2 Higher-Level Control (Directionality)

Even though of crucial importance in a global perspective, the issue of directionality control could not be addressed within the limits of this thesis, and existing HSP systems have bypassed the need for scheduling knowledge. But the way seems clear for incorporating directionality control into HSP, although its viability still requires demonstration.[22] In a pure PSA such as HSP there can be no separate high-level control mechanism -- directionality must be represented as just more "low-level" control. Existing productions can be augmented to make them conditional on various properties of the data matched by the production condition (e.g., validity, or closeness of temporal adjacency match) and on the reliability (strength) of the production itself. Dynamic thresholds used by these added conditions can be represented as separate WM elements. Such methods are used in HSII SASS [Mostow and Hayes-Roth, 1978], and the HSP translations of SASS closely mimic them.

---

[21] However, since very little searching is actually done, it becomes mostly sequential.

[22] Since conflict resolution mechanisms can provide help with directionality control, we might want to add conflict resolution to HSP.

HSII has a sophisticated focussing mechanism [Hayes-Roth and Lesser, 1977] which schedules queues of potential KS instantiations and adjusts dynamic thresholds when appropriate (e.g., to encourage additional activity in a particular region of the utterance). Mostow and Hayes-Roth [1978] argue that proper focussing depends on complex global properties of the working memory, and that such properties cannot be described by the small set of simple conditions in a production. But it may be possible in HSP to represent these complex global properties by WM elements that are constantly updated by special productions monitoring changes to the global state.[23] Then the task productions (some, not all) would be augmented with conditions sensitive to the representation of global state: i.e., they would be self-scheduling. The workability of such a scheme in HSP remains for the time being a source of speculation.

Summary

No separate summary for this chapter is given here. The Representation and Architecture section of the Conclusion (Chapter 7) includes a summary of the main assertions of this chapter.

[23] Though these productions have a special function, they are not to be treated specially by the PSA. They are just mixed in with all the other productions.

Chapter 4

Time Efficiency

This chapter provides an analysis of time efficiency in HSP and a comparison with HSII time efficiency. The bulk of the analysis is based on equivalent HSII and HSP configurations containing only the POM (syllable recognizer) and RPOL (rating policy) KSs, since these two are the only ones completely debugged in their HSP form.[1] The POM + RPOL HSP configuration contains 999 productions, classified as follows: POM (906), RPOL (39), Time propagation (32), and General (22).[2] Only a single run of each system (HSII and HSP) was made, and the input was limited to the segments of a single syllable (the word "DID") from an utterance.[3] This last restriction was partially motivated by the slowness of HSP, but more importantly by the small address problem on C.mmp (see Chapter 6), which limited the total number of elements in WM and the number of changes per PS cycle. These limits could be removed only by reorganizing the HSP system to do more overlaying, and it did not seem worth the substantial extra effort.[4]

We begin with a discussion of several existing efficiency mechanisms in HSP, proceed to a direct comparison of execution time in HSII and HSP, and end with the identification of some remaining sources of inefficiency in HSP. The possibility of exploiting parallelism for time efficiency will not be dealt with here, as it would only complicate matters; the following chapter is devoted entirely to parallelism. Thus, for the time being HSP will be considered a uniprocessor architecture to be compared with the current HSII.[5] HSP's very low multiprocessing overhead (which is, of course, still largely present when running in uniprocessor mode) is what allows us to do this.

[1] Creation of the POM + RPOL HSII configuration for comparison with HSP was not easy. The POM and MOW (word recognizer) KSs are intermingled within a single HSII module, and thus MOW had to be painstakingly excised.

[2] General includes: hyp merging, updating necessitated by new or deleted links, redundant link elimination, and hyp time consistency checking.

[3] In what follows, this test run is often referred to simply as "the POM run", though it is actually POM plus RPOL.

[4] Specifically, subpieces of the WM index and perhaps the lists of current changes would have to be overlayed. WM elements, productions, and subpieces of the PM index are already overlayed.

[5] HSII has evolved (devolved ?) since 1974, the time of Fennell and Lesser's study of parallelism [1977], to become a uniprocessor architecture.

4.1 Existing Efficiency Mechanisms

In order to get HSP off the ground at all it was necessary to incorporate several time efficiency mechanisms. Without them, testing and debugging would have been so slow as to be impracticable. They all have analogues in the HSII system, and some are closely related to efficiency mechanisms in use in other PSAs. Three of the most significant of these efficiency mechanisms are discussed below, including an assessment of the size of their effect.

4.1.1 Production Memory Indexing

HSP contains a pre-compiled index to all productions in its Production Memory (PM).[6] This index takes advantage of the fact that every production begins with a condition element applying explicitly to a WM change: the type of change and identifier of the changed field[7] provide the first two levels of indexing. Beyond that, three additional levels are possible using the first three (presumably fixed) fields of the changed WM element. For example, there Is a POM production for creating legal next syllable states from a newly rated "AL" state, which begins with the two following condition elements:

```
< MOD $ST UVLD >  $ST= < HYP LSEGST "AL" OSEGS/($OS **) >
```

This production is indexed successively through the five levels by `MOD UVLD HYP LSEGST "AL"`. In other cases all five levels of indexing need not necessarily be used. For example, a production which checks for consistency of times on hyps with a changed end time is indexed with only three: `MOD ETIME HYP`.

The function of the PM index is to narrow the number of productions to evaluate in any one cycle from (the total number in PM) times (the number of changes) down to a much smaller number. Its effect is dramatic: in the POM run there were 375 changes, so with 999 productions in PM there would potentially be almost 375,000 productions to evaluate. The actual number evaluated was only 1944, representing a savings of nearly a factor of 200. In general, the PM index reduces a linear dependence of execution time on PM size to a sublinear one. In the POM run there were on average only 5 productions evaluated per WM change. In a larger system, say with 10,000 productions, an average number of 50 might at first be expected. But since new productions tend to respond to different changes than already-existing ones, we can expect instead that the average number evaluated per change will be much less than 50 (and perhaps not much greater than 5).

[6] An early decision was made to rely on hand-compilation of the PM index, to save the effort of building a compiler. This has turned out to be a mistake. Many of the KS bugs that cropped up along the way were due to errors in the manual building of the PM index. In retrospect, building of the compiler looks like a relatively simple task.

[7] This latter is absent for NEW and DEL changes.

The PM index is a filter in the sense used by McDermott, Newell, and Moore [1978] in their investigation of efficiency in the PSA called PSG. The filters discussed there use knowledge from one or more of the following sources: (1) the occurrence of condition elements in productions, (2) which WM elements support which condition elements, and (3) the relationship among condition elements of a single production. HSP's PM index uses source (1), but only for the first two condition elements of each production: the one applying to the change, and the one applying to the changed WM element. Filtering based on information about condition elements after the first two is unnecessary because they do not normally require any WM searching for their evaluation. This is because of the use of explicit WM element references, discussed in the next section.

4.1.2 Explicit Working Memory Element References

In HSP many WM elements contain explicit references to other related elements. This allows productions to locate elements relevant to their operation by simply following references, beginning at the element which triggered the production. For example, a production which wishes to apply some condition to the upper hyp of some link need only bind a variable (say $UH) to the upper hyp reference, and then use the "=" construction to apply a condition element to the upper hyp, as follows:

```
... < LNK ... UHYP/$UH ... > $UH= < ... > ...
```

In many productions, every successive condition element applies to a WM element already located by an explicit reference earlier in the condition evaluation. Such productions require no WM searching at all for their evaluation.

The use of such references is quite extensive overall: of 2214 condition elements encoded in HSP POM, only 2.3% require WM searches for their evaluation. The dynamic behavior gives a similar picture: of 5487 condition element evaluations only 1.2% required WM searching; the remainder were applied directly to a single element located via explicit references.

These explicit references are a natural reflection of HSII and the references that Blackboard elements contain. But it is an unorthodox feature for a PSA, and there are some obvious disadvantages. One of the most serious is the error state that occurs when some element is deleted while other elements still have references to it. The difficulty occurs because the linkage between elements has been made explicit, and hence unconditional. A scheme which relied on repeated searching to establish the linkage would not have this error problem, but would pay with longer (perhaps much longer) execution time.[8]

[8] Perhaps the ultimate solution to this difficulty is to design new hardware architectures that will allow the luxury of large amounts of (seemingly inefficient) searching. After all, PSs are based on an assumption of high recognition-action ratios. So maybe their true benefit will only be realized when new machines free us from petty efficiency concerns.

The savings in execution time provided by explicit references is quite large. It is possible to obtain an estimate of the savings using data from the POM run: the number of condition elements evaluated (+ and -), the total number of matches of condition elements to WM elements, and the size of WM. The calculation was performed for three selected cycles of the POM run. The smallest one (10), a medium one (15), and the largest one (9).[9] The results are shown in Figure 4.1 in terms of how much the use of explicit references speeds up execution time.[10]

	WM size	Factor speedup provided by refs
Cycle 10	83	21
Cycle 15	64	35
Cycle 9	83	52

Figure 4.1 Estimated factor of speedup with use of explicit refs

4.1.3 Working Memory Indexing

In HSII, hyps in the central Blackboard are indexed both by level and by utterance-time region to allow efficient retrieval and searching. Thus it was natural to include a WM indexing mechanism in HSP. The exact mechanism chosen is a two-level association list for WM that groups elements according to the values of their first and second fields.[11] For WM elements which are representations of hyps, this corresponds to HSII's indexing by level; there is no indexing by time region in the current HSP version. For example, in HSP all syllable hyps, which are of the form < HYP SYL ... >, can be immediately obtained from WM by associating first along HYP and then along SYL.

The effect of such a mechanism is to reduce the amount of WM searching necessary to match a condition element. If a condition element begins with one or two fixed fields (which contain constants rather than unbound variables),[12] then the search for a match can be narrowed down considerably. The effect of this could be very large, except that the use of explicit WM element references has already made WM searches relatively rare. Also, it must be remembered that WM size is artificially low in these POM runs. If the

[9] By cycle size we mean, roughly, the number of productions evaluated. A more precise definition appears in the following chapter.

[10] This calculation was performed assuming no WM indexing mechanism (see following section). But obviously the indexing would be needed and would play a much larger role if explicit references were not used.

[11] The resulting requirement that all WM elements begin with two fixed fields is only a minor nuisance.

[12] In practice this is almost always the case.

number of KSs increased by a factor of 5 to 10, and utterance length by 10 to 20, we can anticipate as much as a 100-fold increase in WM size.[13] Under these circumstances WM indexing would be much more effective.

Of the 66 WM searches that occurred in the POM run, none were of the entire WM, 2 were after indexing one level, and 64 were at the second level (and thus most efficient).

HSP's WM index is related to a PSG filter [McDermott, Newell and Moore, 1978] which makes use of knowledge about which condition elements are supported by a WM element (called source (2) above). However, it does not relate WM elements to specific condition elements, but rather classifies WM elements according to conditions commonly tested by condition elements (i.e., the values of the first two fields). The index must be updated when elements are added to or deleted from WM, but at much lower cost than matching against all condition elements as in the related PSG filter.

4.2 Comparing HSII and HSP

An execution time comparison of HSII and HSP is a difficult task due to many differences in the underlying systems, but is nevertheless crucial to a study such as this. There are several avenues for cutting through the inessential differences. None is very satisfactory on its own, but in concert they produce a reasonably believable comparison. In any event, the attempt at comparison is instructive in its own right.

The base data shown in Figure 4.2 is from the single run of the POM + RPOL configuration operating on input segments for the word "DID". The separation of HSP time into POM/RPOL could not be conveniently obtained due to the accounting being tied to individual productions, at which level POM and RPOL are indistinguishable. The data gives a factor of 255 as a starting point for the HSII - HSP comparison.

[13] WM size increase should be roughly linear with utterance length, but not quite so with the number of KSs since activity will stretch out over more cycles.

	HSII (sec)	HSP (sec)	factor
POM PRE	1.30	-	-
POM KS	.75	-	-
RPOL PRE	.76	-	-
overhead	.79	-	-
Total	3.60	917.0	255

Note: HSII took in addition 5.36 sec for initialization. HSP has no analogue.

Figure 4.2 Base execution times for HSII and HSP

Since the time comparison is based on a single, quite small run, there must be something said about its typicality. First, a single syllable is the largest unit which POM operates on, and there is essentially no explicit interaction between syllables within POM (that is relegated to other KSs, notably MOW, the word recognizer). Thus POM's behavior on a total utterance is to a first approximation just the sum of its behaviors on sections of the utterance centered around each syllable nucleus (vowel segment). Next, we can argue that since POM is "table-driven" with respect to the identity of syllables, the same basic pathways are exercised for "DID" as for any other syllable of the same length. The only source of variation still of concern is syllable length. Its effect is unknown, and may be quite large. In fact, we have reason to expect that longer syllables[14] would take proportionately longer in HSP than HSII -- perhaps a half to a full order of magnitude. This is because of the combinatorics of the syllable state transition networks, and the fact that HSII is relatively more efficient in processing these networks.

There is the more serious issue of typicality of POM and RPOL versus other KSs. On the positive side, POM is one of the largest, most varied of all the KSs, and thus embodies a representative collection of KS activities. Yet POM is not typical. Three other KSs (WOSEQ, SASS and MOW with the WIZARD procedure) account for perhaps two-thirds of the total activity in a full HSII run, and they are dramatically different from POM. They have much higher ratios of declarative to procedural knowledge (about 10:1 compared to only 1.2:1 for POM), though this does not directly affect time efficiency. More importantly, these KSs all rely to a greater extent on the efficiency of local working memory and control than does POM. Their activity tends to be more repetitive, i.e., many iterations of a small computational cycle.[15] A direct translation of these KSs into HSP would be a serious

[14] Having two or more consonants on one or both sides of the vowel.

[15] SASS searches a grammar network, MOW-WIZARD searches a state-transition network representation of words, and WOSEQ searches for left and right extensions to word sequences that are both grammatical and highly rated.

mismatch of task to architecture, yielding unbearably large overheads for data-directed invocation and global working memory access.[16] If alternative formulations using more direct recognition (i.e., fewer, larger cycles) could be found for these KSs, then perhaps HSP versions would be viable. Failing that, the time efficiency penalty for HSP would be two orders of magnitude or more.

The HSII - HSP time comp[illegible]n must be normalized. The first avenue for this consists of separate consideration of [illegible]ndividual difference. A numerical factor for each, plus an assumption of independence, giv[illegible] a single overall conversion factor, although one with considerable uncertainty. In the analysis of differences that follows, a factor greater than 1 indicates that HSP is at a disadvantage relative to HSII.

(1) Execution rate of the underlying machine -- MIPS (million instructions per second) rates will be used to normalize for this difference. Fuller [1976] obtained MIPS measurements on a PDP10 (KA10 processor) and C.mmp for a price/performance comparison of the two architectures. A figure of .34 MIPS was obtained for the PDP10 running both a general program mix and a set of four benchmark programs. A figure of about .19 MIPS was obtained for a PDP11/20 processor on C.mmp. Fuller estimated a PDP11/40 to be .34 MIPS (at that time none were yet operational on C.mmp); subsequent experience has shown this estimate to be about 30% too high. (Delays through the C.mmp central memory switch slow the 11/40 down more than expected.) Thus we use a factor of 1.3 for normalizing execution rate between a KA10 and C.mmp 11/40.

(2) Instruction set of the underlying machine -- In spite of its being a minicomputer, the PDP11 has an instruction set which is comparable in power with the PDP10's. Data from Fuller [1976] using four benchmark programs totalling about 3500 instructions gives a PDP11/PDP10 instruction ratio of .9; i.e., fewer instructions were required on the PDP11. Of course, a PDP10 instruction operates on a larger word size, and in some applications (e.g., numerical processing of large integers) this could force a much higher ratio. But speech understanding involves mostly processing of addresses and small integers, so the ratio of .9 should hold true.

(3) Small address problem -- The PDP11 has only 32K 16-bit words of address space compared to 256K 36-bit words on the PDP10. Aside from its drastic effect on design, coding and debugging, the small PDP11 address space mandates execution overhead in the form of overlay swapping and extra copying.

HSP gathers statistics on the number of swaps, and the time for a single swap has been measured, so this effect can be factored out with reasonable certainty. It should be noted that the overhead within L* to reference an overlay dominates the Hydra overhead (which amounts to just changing a relocation register). The latter takes only 200 microseconds,[17] while the total including L* overhead is 1.8 msec.

[16] See the end of Section 4.3.2 for a case in point.

[17] A version also exists in the writeable microstore of the 11/40 C.mmp processors which takes only 20 microseconds. HSP did not use this version.

However, L* optimization could reduce this total considerably. The total swapping overhead is 27 sec, or 2.9% of execution time.

The second form of execution overhead is extra copying of structures from an overlay page to a fixed (non-overlay) page for temporary use, followed by erasure. There are a number of instances of this in HSP, most of them dealing with WM element field values. Accounting code was added to HSP to obtain the number of such copy-erase instances, with the result being 7970 in the POM run. Separate timing of the copy and erase functions yielded a time of 5 to 10 msec for the pair.[18] Thus the copying overhead was 39.9 to 79.7 msec, or 4.4 to 8.7% of total execution time.

Combining the two types of overhead gives 7.3 to 11.6% of execution time, corresponding to a conversion factor of 1.08 to 1.13.

(4) <u>Operating system</u> -- Both HSII and HSP have very little interaction with their respective operating systems during the interval over which they are timed, so this difference can be ignored.

(5) <u>Implementation system (language)</u> -- No good comparative data on SAIL and L* exist, but it is generally assumed that every level of interpretation in a system costs around a factor of 10 in execution time. Experience with LISP compilers generally supports this. It is known that an L* system containing mostly interpreted L* code spends half to two-thirds of its time in the L* interpreter. This fact alone accounts for a factor of 2 to 3. But the difference between SAIL and L* is broader. For example, L* makes heavier use of (relatively) inefficient list structures, and L* does virtually all argument-passing and saving of intermediate results in a central stack while SAIL makes heavier use of machine registers. Since the SAIL-L* difference is rather uncertain, a range of 5 to 10 will be used for a conversion factor.

(6) <u>Degree of kernel optimization</u> -- The version of HSII used has a highly optimized kernel, having gone through a large number of iterations since the first operational version in 1974. A comparison of various primitive operation timings between the Fennell and Lesser system (with locking turned off) and the current HSII show across the board improvements of a factor of 2 to 4. The HSP kernel, on the other hand is still totally unoptimized.[19] Optimization differences in the speech knowledge code (as opposed to the kernel) are minor -- both HSII and HSP POM are only moderately optimized.

[18] The actual times depended on the type, as follows (in msec): integer, 2; WME reference, 8; production reference, 6; list, 50 or more depending on the length. The composite figure of 5 to 10 is based on an estimate of the frequencies of the various types copied.

[19] This was intentional. Not only did it keep the implementation effort within bounds, but more importantly it maintains a realistic ratio of execution time between various system components. E.g., we can rest assured that synchronization overhead in an optimized HSP would be about the same percentage as in the current HSP, because the critical sections have not been selectively optimized.

It should be noted that this kernel optimization category does not include the possibility of optimizing HSP by compiling L* code into machine code -- that difference is covered by the implementation system category above. Examples of optimizations that <u>are</u> included are data restructuring and store-compute tradeoffs.

It is difficult to assign a reliable number, but some improvements similar to those obtained for HSII should also be possible in HSP. Thus a factor of 1 to 3 will be used to correct for the different degrees of optimization in HSII and HSP.

(7) <u>Speech knowledge</u> -- Rather than try to factor out any difference in speech knowledge content (a nearly hopeless task), the HSII and HSP configurations used for comparison were limited to the POM and RPOL knowledge-sources. Since POM and RPOL were translated faithfully from HSII to HSP, the two configurations can be treated as identical in knowledge content. The differences that do exist are insignificant, or else are a reflection of the architectural differences under study.

(8) <u>Complications of parallelism</u> -- As stated at the start of this chapter, HSP multiprocessing overhead is small enough to be ignored in the context of this comparison. What can't be ignored is the complication of the HSP speech knowledge (reflected in longer run times) due to the possibility of asynchronous occurrence of constituent subconditions in some condition being monitored.[20] Often this means that a production must be duplicated: one for each possible change that can make the production true. With an understanding of the structure of HSP POM and data obtained from traces of POM runs, it was possible to obtain an estimate of the execution cost of handling asynchronous satisfaction of subconditions. The result is based on factoring out particular production evaluations and WM element creations/deletions which would be unnecessary in a uniprocess system, giving a factor of 1.05 for equalizing this difference. This factor is much smaller than for the original HSII POM. This is because HSP POM does not really handle the same degree of asynchrony. For example, it assumes that input is instantaneous, i.e., that all input segments are in WM at the start of its execution.

Figure 4.3 brings the individual factors together. There are no known significant dependencies between factors so they may be multiplied to give the total factor of 7 to 42.

[20] The early versions of the HSII POM KS also contained these complications, but most have since been removed because they represent excess generality when running on a uniprocessor.

difference	factor
(1) Execution rate of machine	1.3
(2) Instruction set of machine	.9
(3) Small address problem	1.08-1.13
(4) Operating system	1
(5) Implementation system	5-10
(6) Degree of kernel optimization	1-3
(7) Speech knowledge	1
(8) Complications of parallelism	1.05
Total	7-42

Figure 4.3 Factors for normalizing inessential HSP-HSII differences

A second avenue for comparison covers differences (1)-(6) in one leap by comparing some primitives common to HSII and HSP to obtain an overall conversion factor. Even though the two systems have very different higher level organizations, some low level primitives are similar.[21] For example, both systems read and write elements in a global working memory, differing only in the frequency and pattern of such accesses. Figure 4.4 shows a list of such low level primitive comparisons. To improve comparability, the times for creation and writing do not include the monitoring overhead for data-directed invocation.

primitive	HSP (msec)	HSII (msec)	factor
Create a global WM element	200	3.1	65
Read a global WM element			
Integer value	19	.06	320
List value	21	.6	35
Write a global WM element			
Integer value	18	.15	120
List value	27	2.1	13

Figure 4.4 Timing comparison of HSP and HSII primitives

[21] However, even the low level primitives are affected somewhat by the architectures. For example, the odd fact (seen in Figure 4.4 below) that HSP reading and writing of an integer field take about the same time can be traced to the fact that reading is done by matching a variable to the value to be read. This adds some extra overhead.

These primitives yield a reasonable range of factors, with the exception of the large ones for reading and writing integer-valued fields. The anamoly could be due to those operations being very highly optimized in HSII. In any event, throwing out the large factors gives a range of 13 to 65, which compares reasonably well with 6 to 40 for differences (1)-(6) in Figure 4.3. There is good reason to believe these HSP primitives are less optimal (difference (6)) than the system as a whole. There is one conspicuous inefficiency in both reading and writing that accounts for a factor of 2 alone, so the factor of 1 to 3 used for the whole system may be too small for these primitives. Thus, it is not surprising that the range of 13 to 65 obtained here is somewhat larger than 6 to 40.

The third and final avenue for comparison relies on data from HSC., the C.mmp version of HSII [Lesser and Suslick, 1977], which is also implemented in L*. Although HSC. never became fully operational, the kernel was completed and underwent one optimization pass. HSC. primitives that correspond closely to primitives in the current HSII system were timed, and the results are shown in Figure 4.5. The HSC. timings have locking mechanisms factored out. The resulting HSC./HSII ratios represent a combination of factors (1)-(6). Since factors (1)-(5) are identical for comparisons of HSII with HSP and HSC., removing factor (6) from the HSC./HSII ratios will allow a confirmation of the first avenue of comparison. It is estimated that further optimization of the HSC. primitives (exclusive of L* optimization) could reduce times by a factor of .5 to .8, so we use those values to remove factor (6).

primitive	HSC. (msec)	HSII (msec)	ratio	ratio x .5-.8
Create hyp	225	3.1	73	37-58
Read hyp integer	5.7	.06	[illegible]	48-76
Read hyp list	13.2	.9	15	7-12
Write hyp integer	6.6	1.5	4	2- 4
Write hyp list	15	2.4	6	3- 5
Create link	260	19.1	14	7-11

Figure 4.5 Timing comparison of HSC. and HSII primitives

Taking the lowest and highest points of the individual ranges gives 2 to 76, while the range for (1)-(5) in Figure 4.3 is 6 to 13. This is reasonable agreement, if we consider that the two primitives with abnormally high factors (Create-hyp and Read-hyp-integer) are more highly optimized in HSII than the others. Excluding those two, the range from Figure 4.5 is 2 to 12.

The results of the three different avenues are consistent enough that we can have a fair degree of faith in the estimate of 7 to 42. Applying this as a correction to the base factor of 255 from Figure 4.2 produces the statement that the current HSP architecture is inherently 6 to 36 times slower than HSII for the POM + RPOL configuration. Projecting to a normal mix of syllable lengths for POM input adds perhaps a factor of 2 (i.e., long syllables are relatively uncommon). And projecting to a more complete KS configuration, inexact though it is, adds another one to one-and-a-half orders of magnitude on top of this, giving roughly two to three-and-a-half orders of magnitude total (i.e., 100 to 3000).

We might justifiably place most of the blame for this large difference onto the conventional machine architectures we are forced to use to implement PSAs. The ideal machine for a PSA would combine extremely high parallelism for fast recognition with a huge memory to accommodate vast numbers of complex productions. However, given the speculative nature of such a machine, it is best to return to the question of HSP running on conventional machines. The following chapter does address the possibility of running HSP on medium-scale multiprocessors (10 to 50 processors).

4.3 Remaining Sources of Inefficiency in HSP

Having uncovered this rather large inherent time inefficiency of HSP, the next immediate concern is to account for it: first to understand what in the structure of HSP makes it so much slower than HSII, and then to gauge whether there is any hope for improvement.

To begin, let us look at the global breakdown of execution time in HSP. Figure 4.6 shows such a breakdown into six categories for the complete POM run.

	% total execution time	
Recognition	95	
Production Memory indexing		7
Successful condition evaluation		21
Action interpretation		7
Unsuccessful condition evaluation		60
Action	5	
Deletion of previous change elements		2
Making current changes		3

Figure 4.6 Global breakdown of HSP execution time

There is nothing terribly surprising about this data. It might perhaps be said that unsuccessful condition evaluation takes too large a share, but it is unreasonable to expect its share to be very small, as if the system were non-deterministic. Besides, reducing the 60% contribution of unsuccessful condition evaluation by an order of magnitude would give only an overall factor of 2 speedup.

Although not apparent from the data above, there are several identifiable sources of inefficiency in HSP. We turn now to a discussion of them, including where possible an estimate of their magnitude.

4.3.1 High Degree of Data-directed Control

It was stated in the previous chapter that HSP uses almost 500 times as much data-directed control as HSII (by measure of the number of data-directed invocations). Although that is a positive aspect in some regards, the overhead of this large number of WM-mediated control links is very worrisome. The sequencing of control within an HSII KS has essentially negligible overhead since it amounts to the sequencing of the underlying machine. When this same KS is represented as productions in HSP, the sequentiality of control becomes riddled with data-directed invocations. It is not clear that this is an inevitable result of any PSA. It is possible to imagine much larger productions than in HSP, with considerably fewer data-directed invocations. But an unmanageable explosion in the number of productions seems to go hand in hand with making productions larger.[22]

In many cases a data-directed invocation is triggered by some permanent change to WM, which would have occurred whether or not the resulting invocation was desired.[23] Other cases which cannot be tied to some naturally occurring WM change require a special temporary WM element (often called a <u>pure control</u> signal) to trigger the invocation, resulting in extra overhead for the WM element creation and deletion.[24] The overhead for a single invocation in HSP is about 350 msec for a "natural" one, or 600 msec for a signalled one. By way of comparison, these times are 20-30 times larger than the basic HSP WM read time, and are of the same order as the time to evaluate a production.

In the HSP POM system, virtually all data-directed invocations could be eliminated by making each production, based on the nature of the WM changes in its action, directly invoke the set of productions that respond to those changes. Sufficient information to do this exists at production compilation time; in fact, it is the same information that is compiled into the PM index (and then used interpretively at run time). This would produce a system that uses local control exclusively, yet still uses the global WM as in the current HSP. Of course openness and flexibility would be destroyed (which is why we did not actually do it), but the fact that removal of data-directed invocations is possible makes it proper to view them as overhead.

The overhead consists of all the PM indexing time, change element creation and

[22] Since internal disjunctions are not permitted in HSP, alternative subconditions cause a multiplicative increase in the number of productions. Permitting disjunctions would solve this but lose most of the parallelism, besides being aesthetically unpleasant.

[23] This is a tricky point. Presumably such a "permanent" change is needed in the context of some future computation or by some other KS.

[24] A pure control signal can also be a new field added to an existing WM element rather than a whole new element. Such cases were negligible in the POM run.

deletion, plus the time to create and delete pure control signals in WM.[25] Another less obvious overhead is the initial portion of condition evaluation for each production which is merely reobtaining the context established by the preceeding production (i.e., reading from WM into local variables). A direct invocation scheme could pass the context as parameters to the following production, with much lower costs. The cost of this in HSP is hard to estimate accurately, but we expect that on average about 20-40% of condition evaluation costs are for reobtaining context. Figure 4.7 shows estimated costs of data-directed control in the HSP POM run as a percentage of total execution time.[26] The data shows that we could expect a factor of 1.4 to 1.8 speedup in converting HSP largely to local control.

	% total execution time
Creation/deletion of change elements	6
Creation/deletion of control signals	2
PM indexing (changes => productions)	7
Condition evaluation to reobtain context	15-30
Total	30-45

Figure 4.7 Overhead for data-directed invocation in the HSP POM run

Curiously, overhead for data-directed invocation in HSII is not smaller in proportion to the 500 times fewer invocations: it amounts to about 9% in the HSII POM run. This is because the overhead is dominated by the monitoring of Blackboard changes (of which there were about 200), not by the actual invocations.

4.3.2 Limited Local Working Memory

Given that an HSP KS must use global WM for many functions that a corresponding HSII KS would perform in local memory, it is of interest to ask what this costs HSP in time efficiency. The obvious extra costs of limited local working memory in HSP are due to the relative slowness of reads, writes, creations, and deletions of WM elements that correspond to local operations in HSII. Figure 4.8, which shows numbers of global working memory operations in the corresponding POM runs, gives an idea of the scope of these extra costs.

[25] The control signals are often also used for "passing parameters" to the subsequent production, but we assume that a scheme for direct (as opposed to data-directed) invocation could pass parameters with negligible overhead.

[26] Cases of subroutine control are included in this analysis since they are accomplished in HSP by two separate data-directed invocations ("call" and "return").

	HSII	HSP
Reads	2160	12100
Writes	270	210
Creates	50	110
Deletes	0	60

Figure 4.8 Total of global working memory operations in the HSII and HSP POM runs

This data serves only crudely for this purpose because there are a number of other possible reasons for differences in these counts. For example, some of the extra HSP reads are due to redundant condition evaluation (see Section 4.3.5). And we would expect HSP to have more writes than HSII, but since the opposite is true there must be another factor at work here also.[27] However, we do see more creations/deletions and many more reads in HSP, and the bulk of this must be due to HSP's limited local working memory.

Figure 4.9 gives estimated time factors for global versus local working memory operations in HSP. The local operations are of course hypothetical, and are based on times for simple analogous operations in HSP's implementation language. Creation is problematical since it is not clear what the local correspondent should be; what we use is the time to create and initialize a small (about 5 element) vector. The values for creation, writing and deletion, unlike those in Figure 4.4, include the part of their cost related to data-directed invocation: the time to create and delete a change element (about 100 and 50 msec, respectively), plus the time to index into the PM index with the change (100 msec for the POM + RPOL configuration).[28]

[27] One further note: the fact that the number of creates minus deletes is exactly the same for both systems is largely coincidental.

[28] One possibility for reducing these costs is a feature for marking some WM elements as unmonitored so that changes to them will not be responded to. HSII does have this feature, even though it is needed less there (because data that does not need to be monitored will usually be stored locally rather than in the HSII Blackboard).

	global (msec)	local (msec)	factor
Create	400	4	100
Read	20	.5	50
Write	300	.5	600
Delete	260	2	100

Figure 4.9 Rough factors for global vs. local working memory operations in HSP

If we combine data from Figures 4.8 and 4.9, we obtain percentages of total execution time for each of the WM operations, as shown below in Figure 4.10. Assuming for the moment that most of these operations could be replaced by corresponding local operations,[29] their contribution to total time would essentially vanish because of the large factors in Figure 4.9.

	time (msec)	# in POM run	total time (sec)	% time
Create	400	110	44	5
Read	20	12100	242	26
Write	300	210	63	7
Delete	260	60	16	2
Total			365	40

Figure 4.10 Time contribution of HSP WM operations in the POM run

However, it makes no sense to convert from global to local working memory use without also converting to local control, since data-directed (global) control is built upon the global working memory. Thus we must consider the effect of going from global to local working memory use in an HSP system which has already converted to local control (as discussed in the previous section). This is more difficult to estimate. The global create, write and delete would be less expensive than above because of no monitoring for data-directed invocation. There would be no deletes and fewer creates because no control signals are necessary. There would be an estimated 25% fewer reads because of no condition evaluation to reobtain context. And the total time would be reduced by 30-45% from 917 sec to about 570 sec. Figure 4.11 shows 40% as the resulting estimate of

[29] Some few would have to remain to provide inter-KS communication.

percentage of total time for WM operations in this hypothetical system. Thus we can expect a speedup factor of 1.7 in converting an HSP system which already uses local control from global to local working memory use.

	time (msec)	# in POM run	total time (sec)	% time
Create	300	50	15	3
Read	20	9000	180	30
Write	200	210	40	7
Delete	160	0	0	0
Total			235	40

Figure 4.11 Time contribution of local-control-HSP WM operations in the POM run

There is another less obvious cost of HSP's limited local working memory. Some WM changes that correspond to local HSII operations cause extra productions to be evaluated when they are obviously (to the observer, not the system) irrelevant. The exact size of this cost is unknown, but it is probably well below 10% of total execution time.

An interesting piece of HSII history indicates the efficiencies possible through HSII's use of local control and working memory. The first versions of the SASS KS used the Blackboard to store all intermediate grammatical structures, and were found to be unbearably slow. A reimplementation of SASS using local data structures rather than the Blackboard was able to do much more processing and yet run significantly faster [Hayes-Roth, Mostow and Fox, 1977]. It has been estimated that this new SASS runs two orders of magnitude faster than the original Blackboard-based version [Lesser and Erman, 1977]. The new SASS has three main activities: creation of nodes representing partial phrase hypotheses, searching for already-existing nodes satisfying certain properties, and selecting a next node to process. Figure 4.12 compares how long these operations take in the actual SASS (using local structures) with an estimate of how long they would take if SASS used the Blackboard.[30] Creation in the local version maps into creation of one hyp and a couple of links in the hypothetical Blackboard version, local reading into an associative retrieval from the Blackboard, and local control sequencing into a data-directed invocation.

[30] This is not exactly the same as comparing the new and old versions of SASS, though the old version does use the Blackt [illegible]

	global (msec)	local (msec)	factor
Creation	40	.1	400
Searching	100	.3	300
Control sequencing	100	.03	3000

Figure 4.12 Factors for global vs. local operations in the HSII SASS KS

These factors are so large that it is not difficult to see how an overall factor of two orders of magnitude or more could arise. It is not known exactly what proportion of SASS's total execution time is covered by the above three activities, but it is certainly substantial. And, for example, if the proportion were 50% for the three activities together, an overall factor of well over two orders of magnitude could easily result.

4.3.3 No Declarative Long-term Knowledge

As discussed in the previous chapter, the absence of a declarative long-term memory in HSP leads to a number of awkward procedural representations. Since these usually involve large sets of mutually-exclusive productions which all respond to a very similar WM stimulus, there is potential for a large amount of unsuccessful production evaluation. Thus there is a marked effect on time efficiency as well as space efficiency.

The time efficiency of these large mutually-exclusive production sets is very dependent on the PM indexing mechanism. Indexing is sufficient in most cases to select the single production to evaluate, essentially nullifying any time efficiency loss. There are a few exceptions, the most notable being the POM state transition probability productions, where indexing narrows the number of productions to evaluate from 589 down to about 5 or 10. In any case, absence of the PM indexing mechanism would have disastrous consequences for the time efficiency of these large production sets.

Details of the POM run were analyzed to estimate the time cost of the lack of declarative long-term memory. In the cases where PM indexing was sufficient to select a single production, no time penalty was assigned, since that selection is about as efficient as a true declarative memory structure would allow.[31] In the remaining cases, the cost of all extra unsuccessful production evaluations (and in the case of the state transition probabilities, extra WM element creations and deletions) was tallied. The result is that 15% of the total execution time is inefficiency introduced by the lack of a declarative long-term memory facility.

[31] Note that the PM indexing mechanism is being used in a peculiar way as a declarative memory structure. But since it is purely an efficiency mechanism and is transparent to the representation, the claim of no declarative long-term knowledge has not been violated.

4.3.4 Working Memory Searching

As discussed earlier, the existing HSP mechanisms of explicit WM element references and WM indexing drastically reduce the amount of WM searching. But the prospect of large increases in WM size that come with more KSs and longer portions of input utterances indicates that WM searching is still a problem. As noted earlier, 100-fold increases in WM size are quite possible.

It was not convenient to directly measure in HSP the fraction of time spent searching WM, but it is possible to get an estimate from other data taken from the POM run. There were a total of 1803 matches of condition elements to WM elements which failed; subtracting the number of productions which failed (since there must be exactly one match failure per production failure, exclusive of WM searching) gives 219 negative matches as the cost of WM searching. Since matching is the dominant cost of condition evaluation, and assuming all matches cost about the same, the ratio of 219 to the total number of matches, 5691, gives the cost of condition evaluation due to WM searching as 3.8% of 81%, or only about 3% of total execution time.

It is possible to get a rough estimate of how WM searching costs will increase with larger system configurations. The addition of new KSs and accompanying Blackboard levels (as represented in the HSP WM) will probably not increase the proportion of WM searching time, because new indexing structure will be also added to the WM index for the new levels, with searching normally necessary only within a single level. However, since there is no indexing of WM by time interval, an increase of 10 to 20 in utterance length will increase the time for most searches by a similar factor (this in addition to a 10 to 20-fold increase in the number of productions evaluated, and hence the number of searches). Thus, as shown in Figure 4.13 below, we can expect WM searching to increase from an insignificant 3% of total time to a considerable 30%. (This assumes a factor of 15 increase in utterance length).

		% total execution time	
Current HSP POM run			
	PM indexing	7	
	Condition	81	
	Searching		3
	Remainder		78
	Action interpretation	7	
	Action	5	
Projected HSP run with larger configuration			
	PM indexing	5	
	Condition	87	
	Searching		32
	Remainder		55
	Action interpretation	5	
	Action	4	

Figure 4.13 Proportionate cost of HSP WM searching

Searching in HSII is done both by kernel functions (e.g., finding a hyp to match given characteristics, or finding hyps adjacent in utterance-time to a given one), and by the KS code itself (e.g., using a FOREACH set iteration statement). It is not feasible to measure the contribution of the latter, but in the HSII POM run there were 14 retrieval calls, amounting to 3% of total execution time.

4.3.5 Redundant Condition Evaluation

In a paper describing the efficiency mechanisms of the OPS system, Forgy [1977] identifies two aspects of redundant condition evaluation. The first, structural redundancy, arises from the fact that some condition elements appear in many different productions, and thus may be re-evaluated every time they are encountered. The second, temporal redundancy, is caused by condition elements being matched against the same WM elements cycle after cycle even though the WM elements have not changed. These two types of redundancy are exactly those addressed by the first two sources of knowledge used in PSG filters (as cited above in Section 4.1.1).

The HSP PM index already serves to reduce structural redundancy, and the WM index reduces temporal redundancy. Furthermore, HSP's explicit WM element references reduce WM searching to such a low level that evaluating a redundant condition element is usually not very costly. Thus, there is not a large factor still to be gained in HSP through redundancy elimination. An additional factor of 2 or perhaps 3 is all we should hope for.

There are local cases where the redundancies are still worrisome. But often in these cases there are other remedies, such as the following two examples which use intermediate storage in WM.

In POM, the state transition probability productions require five pieces of context for their operation. A straightforward implementation of these productions requires an initial six condition elements to gather the context, and since many of these productions must be evaluated for every relevant change (though only one is true) there is a significant amount of structural redundancy.[32] A savings was realized by restructuring the productions to first gather the context and create a WM element to hold it, then on the following cycle react to the new WM element to re-obtain the context and produce the result. The fact that the context has been packaged into a single WM element allows the PM index to discriminate on the context information to eliminate most of the false productions.

For linking up the context of segments around a syllable nucleus segment there are two sets of productions, due to the complications of parallelism as discussed in Section 4.2. They can be schematized as follows:

```
Pa : ( chga A B C D E F G -> ... )
Pb : ( chgb E D C B A F G -> ... )
```

where **chga** and **chgb** are condition elements which test for a change which makes, respectively, condition elements A and E true. **Pb** is needed because **Pa** may fail at E, but E may later become satisfiable, in which case **Pa** can no longer fire because it is conditional on **chga**, i.e., some change that makes **A** true. The re-evaluation of A, B, C, and D that Pb does is a form of temporal redundancy. If we know that **A B C D** will not in the meantime become false, we may add **Pa'** and redefine **Pb** as follows:

```
Pa' : ( chga A B C D  NOT E -> X )
Pb : ( chgb E X F G -> ... <DEL X> )
```

This avoids the redundancy. The purpose of X is to store in WM the fact that **A B C D** is true.

Summary

A factor of 6 to 36 is the inherent speed advantage of HSII over HSP for the POM run. The several sources of inefficiency discussed above contribute toward this factor as shown in Figure 4.14. Factors (1) and (2) are independent since (2) was estimated in a hypothetical system which had (1) already applied. (3) and (4) are not independent of (1) and (2), but their effects are small enough that the dependence makes little difference to the combined factor. (5) has a large overlap with (2), so the estimate for (5) has been reduced somewhat, but it is just a rough guess anyway.

[32] This is true even though none of the six condition elements requires searching.

source	factor
(1) Data-directed control	1.7
(2) Limited working memory	1.7
(3) No declarative long-term knowledge	1.2
(4) WM searching	1.03
(5) Redundant condition evaluation	2
Total	7

Figure 4.14 Sources of HSP inefficiency with factors of slowdown

The combined factor of 7 reaches the lower bound of the 6 to 36 range we are attempting to account for. Without coming closer to the middle of that range, there is some reasonable doubt that all the major sources of inefficiency have been captured. There are indeed several known sources not discussed here, mostly connected with the limited capabilities of the HSP production language. For example, the inability of a single production to deal with data lists of arbitrary length is known to have a significant effect in the POM and RPOL KSs. Perhaps such additional sources have a larger effect than anticipated.

A section on time efficiency in Chapter 7 summarizes the major assertions of this chapter.

Chapter 5

Parallelism

The task of understanding speech in real time requires a great deal of computing power, and trends in computer technology make multiprocessor machines such as C.mmp [Wulf and Bell, 1972] and CM* [Swan, Fuller and Siewiorek, 1977] an attractive solution. But parallel machines such as these can be effective only if first there is enough parallelism inherent in the task, and secondly the system architecture can exploit it. Lesser [1974] recognized the great importance of parallelism for speech understanding, and provided a survey of design issues for appropriate system architectures.

In 1974 Fennell and Lesser built a multiprocessor simulation within the HSII system (itself running on a uniprocessor PDP-10) to determine the degree of parallelism, and to study interference in Blackboard access [Fennell and Lesser, 1977], [Fennell, 1975]. Even though it was not possible to match in HSP the knowledge source (KS) configuration they used (most of those KSs were phased out of HSII by 1976), their data provides a useful comparison point for HSP in the analysis that follows.

This chapter examines the way in which HSP exploits parallelism in the speech task, and compares HSP with the parallel (Fennell and Lesser) version of HSII. It uses data from multiprocessor runs of HSP on C.mmp and from an HSP simulator which can simulate even larger numbers of processors. The first section describes where potential parallelism exists in the speech task, and compares HSII and HSP in the way they exploit this potential. The next two sections describe the general methodology that was necessary to obtain HSP timings on C.mmp, and the HSP simulator. Next come two sections presenting the results on HSP parallelism and comparison with HSII: one on the overhead for exploiting parallelism, one on the degree of parallelism and limits to greater parallelism. The final section on hardware memory interference is a side excursion into an interesting problem which HSP experienced on C.mmp.

5.1 The Sources of Parallelism

There are several sources of parallelism in the HSII architecture, corresponding to the three-dimensional structure of the HSII Blackboard:

(1) <u>Different</u> <u>information</u> <u>levels</u> -- The speech KSs that operate at the different Blackboard levels can in some cases be executed in parallel. It might be expected that the parallelism is as large as the number of KSs. But this is actually quite unlikely, since KSs tend to be directly dependent on the results of others operating at nearby levels. For example, a totally bottom-up system would have no such

parallelism, while a pure combination bottom-up and top-down system would have a potential parallelism of this kind of only two.

(2) Different intervals of the utterance-time -- Information units in the Blackboard span intervals of utterance time, and units that are non-overlapping can often be processed in parallel. The lowest level of the Blackboard (segmental) has the smallest units and hence the most parallelism of this type; the highest level (phrasal) has very little since each unit spans much of the utterance.

(3) Alternative hypotheses -- The errorful nature of speech knowledge requires that many alternatives be maintained in the Blackboard for a given time within a given level. This creates a combinatorial searching problem, but the processing of these alternatives can be done in parallel. The higher levels of the Blackboard contain more alternatives than lower levels due to combinatorial propagation of uncertainty.

The HSP architecture exploits these same three sources, but introduces an additional source:

(4) Intra-KS parallelism -- KSs, which are single units in HSII, have in HSP a fine structure with productions as units. For example, the HSP POM KS has 906 productions, resulting in some intra-KS parallelism which is not exploited in HSII.[1] The parallelism is not nearly on the order of the number of productions. But it is larger than the number of true parallel intra-KS activities;[2] it includes a large dose of parallel evaluation of conditions which turn out to be false.

Neither HSII nor HSP fully exploit these various sources of parallelism. For example, in HSP it is possible that a production may find multiple matches for one or more of its condition elements, and in this case the alternatives are processed sequentially by the single production instantiation.[3] In HSII, a single KS instantiation can similarly process a number of alternatives sequentially. Often this takes the form of retrieving a set of Blackboard elements according to some criteria, and then processing them sequentially.

[1] Furthermore, the totally data-directed nature of control in HSP allows this extra parallelism to be had almost "for free". No explicit operations such as FORKs and JOINs are required. Control within the HSII KSs is not data-directed, so that exploiting this same parallelism there would be more difficult.

[2] An example of this in POM is the parallel recognition of the left and right halves of a syllable.

[3] It would be possible, though difficult, to modify the HSP architecture to process these multiple instantiations in parallel. However, these cases of multiple firings of a single production may be infrequent. There were none in the POM run. But some could be expected with a richer KS set and larger portion of input utterance.

Parallelism in the HSII Control Cycle

In HSII, the determination of which Preconditions (PREs) should be invoked in response to a Blackboard change is done as a subroutine of the kernel function that actually makes the change. This is based on a static specification for each PRE in the system of which fields of hyps or links, and in which Blackboard level, the PRE is monitoring. As a result of a Blackboard change, the PREs that are interested in that change are invoked in parallel. Each PRE instantiation may in turn invoke KS instantiations in parallel, but each of these will run to completion without further splitting. The cycle is completed when the KSs themselves make changes to the Blackboard that trigger other PREs. Note that the PRE and KS invocations occur in a totally asynchronous manner.

Parallelism in the HSP Control Cycle

The HSP control cycle differs from that of HSII in several important ways. The asynchronous aspect of invocation is replaced by a global synchronization imposed by the PS cycle. Changes to WM are not made during the firing of a production, but are queued and then made at the end of the cycle. And there is no duality analogous to HSII's PRE and KS; a production cannot directly invoke another, but can only make WM changes which may result in the invocation of certain other productions on the following cycle.[4]

A cycle begins with a queue of the changes that were made to WM in the previous cycle (the <u>change queue</u>), an empty queue of productions to be evaluated (the <u>production queue</u>), and a single active processor. The active processor immediately signals all others, and all processors begin the recognition phase. Each processor enters a loop in which it repeatedly evaluates a production from the production queue, or if none exists then processes a change from the change queue.[5] Processing a change consists merely of indexing into PM and putting onto the production queue (for subsequent evaluation) all the productions so obtained.

Whenever a production evaluation succeeds (the condition of the production is True), its action is interpreted to obtain symbolic representations of the changes, which are accumulated in a temporary memory. (The changes are not actually made at this time, thus the use of the term "interpretation").

Each processor passivates itself as soon as it finds both queues to be empty; except for the last processor to finish, which enters the action (or change execution) phase. In this sequential stage the single processor first erases the change elements created during the previous cycle. It then makes all the accumulated WM changes of the current cycle,

[4] These linkages can be very explicit, with the WM change being nothing more than an invocation signal for another specific production. The point is that the signalling is still done in a data-directed manner through the WM.

[5] Note that change processing and production evaluation are intermixed. This is done to prevent the production queue from growing too large. It makes no difference to the results.

and puts the change elements representing these changes into the change queue in preparation for the following cycle.

Since it is possible that one or more processors may passivate themselves while another is still processing the last change (which may produce additional productions to evaluate), there must be a provision for reactivation of processors when that last change results in new production queue entries. Thus, with every insertion into the production queue a check is made to see if any processors are idle, and if one is found it is signalled to start again.

There are mutual exclusion semaphores to protect access to the production and change queues, the list of accumulated changes, and the count of passivated processors. The latter is used to detect the end of the recognition phase, and also in the above-mentioned reactivation check.

Synchronization of available space lists in shared memory is also necessary. The representations of the changes obtained by interpreting actions must go into the change queue to drive the next cycle, and therefore must be created in shared memory so that they are accessible by any of the processors. In addition, new WM elements are created during action interpretation, and they must of course also be in shared memory. Thus action interpretation is protected by a mutual exclusion semaphore.

5.2 Timing Methodology

There were a number of difficulties presented by C.mmp/Hydra for obtaining accurate, consistent timings. It was possible to control many of them by various means, but a few were insurmountable. The net result is that timings are accurate only to 5 or 10%. Much of the special methodology for controlling timing runs is interesting because it reveals in an oblique way some important aspects of the C.mmp/Hydra architecture. A partial listing follows:

> Two different types of primary memory are in use on C.mmp. About 40% of the total is semiconductor memory, and core memory makes up the remainder. In the absence of memory contention the core memory seems to be roughly 10% faster, but as contention increases the core degrades more quickly. Thus no single correction factor would suffice. It was hoped that HSP could be run with only core memory, but it turned out to be insufficient. Even with only one-fourth of the semiconductor memory removed from the system there was occasional paging. Also, fewer ports aggravated the memory interference problem (see Section 5.6). It was possible to largely correct for the different memories by normalizing times for each run so that action times are equal (required by the architectural feature of sequential action execution). The 4 process run required a 10% correction, and the 7 and 10 process runs required about 5%.
>
> The timing facility provided by Hydra does not deliver pure user execution time; it

includes overheads such as scheduling, paging, and interrupt service for I/O devices. However, these overheads were eliminated or minimized by various means: running alone on the system, running with enough primary memory to eliminate paging, locking HSP processes to disjoint processors, locking the processes into core to ensure that they are not paged out, giving HSP processes the equivalent of an infinite time slice to eliminate scheduling overhead, and avoiding whenever possible the use of processors having high speed I/O devices.

The five 11/20 processors each run at .65 times the speed of an 11/40 on C.mmp. This difference could not be factored out satisfactorily for multiprocessor runs, so only 11/40s were used. Even 11/40 processors run at slightly different speeds; however, these differences are small, typically less than 1%.

The total HSP system is quite large, and under normal circumstances (i.e., with other users on C.mmp) it is unwise to require that all of it reside in core at once, so some of the pages are swapped in and out of the core page set (CPS) as needed. But to eliminate this overhead for timing runs, all needed pages are permanently loaded into the CPS.

The timing version of HSP uses busy-wait synchronization for all critical sections and process signalling. Synchronization mechanisms provided by Hydra have unacceptable overheads, especially for large numbers of processes.[6]

Small differences can be magnified due to a reordering of close events, resulting for example in a different pattern of critical section interference.

5.3 The HSP Simulator

Since parallelism in HSP was anticipated to be higher than that attainable on C.mmp (with a maximum of 16 processors), a simulation subsystem was built within HSP to explore the consequences of a very large number of processors; i.e., 50 to 100.[7] The system can collect timing data from a uniprocessor HSP run on C.mmp, and then use that data to simulate runs with any number of processors.

The 1 msec grain of the simulation is detailed enough to include the short critical

[6] Hydra policy system (version 0) semaphores (the standard user-level synchronization mechanism), when used to signal a set of processes, take 90 msec for the first and 50 additional for each succeeding process. (This was improved in version 1 of the Policy Module, which allows a concurrency of three for such signalling.)

[7] This number of processors is on the order of that eventually planned for the CM* multiprocessor being built at CMU [Swan, Fuller and Siewiorek, 1977], so the simulations should be relevant to running HSP on CM* in the future.

sections involved in accessing the change queue and production queue,[8] and to account for quiescence of the multiple processors at the end of a cycle. It does not include the effect of hardware memory interference -- this phenomenon will be discussed in Section 5.6. Real runs on C.mmp with up to 10 processors showed almost negligible interference due to the critical sections for queue accessing (see Figure 5.10). But the simulation of these critical sections was included in case interference would become significant with a very large number of processors (it didn't)

The simulator was validated by comparison with a series of multiprocessor runs on C.mmp. Two selected PS cycles from the total of 21 in the POM run were used: a medium one (15), and large one (9). These were chosen because Cycle 9 typically had the highest parallelism, and 15 was intermediate between 9 and the cycle with lowest parallelism.[9] Their characteristics are shown in Figure 5.1.

	Cycle 15 medium	Cycle 9 large
# changes responded to	16	36
# productions evaluated	110	300
# productions fired	18	42
# new changes	18	42
uniprocessor time (sec)	47.5	99.3

Figure 5.1 Characteristics of the POM cycles used for validation

The results were compared on the basis of elapsed time, idle time, and the total blocked time attributable to each critical section. (These various sources of lost time are explained more fully in Section 5.5). Figure 5.2 shows the results obtained.

[8] Though only barely so. See difficulty (4) later in this section.

[9] Runs of more cycles were deemed unnecessary because cycle size alone is such a major determinant of parallelism.

category	Cycle 15		Cycle 9	
# of processors	Real	Sim	Real	Sim
Elapsed time				
1	47.5	48.3	99.3	99.1
2	24.9	25.4	50.7	52.5
4	13.4	14.1	26.2	26.2
7	9.3	9.2	16.4	15.8
10	7.4	7.7	12.3	11.7
Idle during recognition				
1	0	0	0	0
2	.12	.26	.16	.01
4	.35	1.30	.58	.71
7	2.97	2.85	.97	.88
10	2.90	7.99	1.25	2.30
Blocked on action interpretation				
1	0	0	0	0
2	0	.14	.13	.11
4	.05	.46	.36	.29
7	.34	1.01	.98	.81
10	1.77	1.69	1.70	.86
Blocked on change queue				
1	0	0	0	0
2	0	.001	0	0
4	.001	.01	.001	0
7	.01	0	.01	0
10	.02	0	.02	0
Blocked on production queue				
1	0	0	0	0
2	.002	0	.01	.002
4	.01	0	.02	.02
7	.02	.03	.03	.04
10	.03	.06	.05	.10

Figure 5.2 Validation of the HSP simulator (times in seconds)

There are several reasons why exact agreement should not be expected:

(1) All the difficulties with consistent timings discussed above affect the agreement since the simulator works from one set of timings and its output is compared with a different set.

(2) The simulator doesn't include a small amount of overhead in the real system for accounting facility hooks. (Those overheads exist even though the accounting facilities are turned off during timing runs).

(3) The simulator does not model hardware memory interference. In spite of special efforts to eliminate interference in the real runs, it was not possible to do so entirely.

(4) The queue accessing critical sections are too short to be accurately represented given the 1 msec grain size of the simulator. Also, the length of a critical section for removing from a queue varies depending on whether or not the queue is empty, but the simulator treats it as constant. The simulator uses 1 msec for all queue-accessing critical sections, while the actual values range from .4 to 1.4 msec.

Taking into account the reasons for expecting discrepancies, the agreement is quite good. Total times are typically 1 to 5% high for for the real run, probably due to (2). The idle and blocked times are not so close as the total times, but that is not surprising in light of (4). Thus the simulator cannot be used as an accurate predictor of idle and blocked time.

The principal use of the simulator for the results of this thesis was in estimating parallelism with more processors than exist on C.mmp (see Section 5.5). Another possible use was in estimating parallelism with larger cycles, as would be expected with more KSs or longer speech input. This turned out to be infeasible since the simulator runs too slowly to simulate large numbers of processors, even with the relatively small cycles of the POM run. For example, Cycle 9 (the largest) can barely be simulated for 50 processors. But since it gets a full 60% utilization during the recognition phase, cycles much larger than Cycle 9 can be expected to saturate 50 processors (i.e., get close to 100% utilization during recognition). Thus with large cycles more processors than 50 must be simulated to probe the limits of parallelism.

5.4 Multiprocessing Overhead

Any software architecture that aims to exploit parallelism in some task domain must supply mechanisms for multiprocessing: process creation, scheduling, intercommunication, and synchronization. But architectures may differ in the amount of overhead required by these mechanisms. Indeed, as we shall see below, HSII and HSP are strongly distinguished by the amount of multiprocessing overhead they contain.

In the HSII architecture there is a necessity for Blackboard locking mechanisms to maintain data integrity when KS instantiations can simultaneously access the same region of the Blackboard; i.e., the same utterance-time interval at a particular level. This produces a strong tradeoff between KS execution interference in the Blackboard (high with simple locking mechanisms) and locking overhead (high with sophisticated locking mechanisms designed to reduce interference). The locking structure used in the Fennell and Lesser system has a dual aspect: whole regions can be locked, or individual hypotheses and links. The two aspects are coordinated; e.g., locking a time region effectively locks all hyps and links whose specified time interval overlaps the region. The region lock involves less overhead than the individual lock, but produces more execution interference since more hyps and links are affected. A figure of 27% overhead for Blackboard synchronization, largely independent of the number of processors, is quoted by Fennell and Lesser.[10]

In contrast to HSII, HSP has no locking mechanism in the global working memory. This is possible because of three properties of HSP, all of which are true of PSs in general:

> Explicit separation of read activity from write activity in the knowledge representation. Productions are all separated into condition and action; condition evaluation is a working memory read-only process, action execution is write-only.
>
> Global synchronization of the recognize-act cycle. Since there is one central recognize-act cycle for the entire set of productions in the system, and since all active knowledge in the system is encoded as productions, it is guaranteed that reading of working memory will never occur simultaneously with writing.
>
> High recognize-act ratio.[11] This allows actions to be executed sequentially in HSP without a severe loss of parallelism (although it becomes severe with very large numbers of processors: see Section 5.5). And sequential action execution means no synchronization is necessary to prevent simultaneous modification actions from destroying a working memory structure.

---

[10] This overhead figure does not include the lost time when a processor is blocked due to a Blackboard lock. Lost time is analyzed in Section 5.5.

[11] Note that what we have called action "interpretation" is here included under the "recognize" part. The "act" part is just the change execution phase, where WM changes are actually made.

The synchronization that is necessary in HSP (for change and production queues, action interpretation, and action execution) is all accomplished with low-level semaphore operations having negligible overhead.[12] However, each call on a semaphore operation does take a couple of L* interpretation cycles, at a cost of about .15 msec per cycle. In POM cycle 2, for example, this amounted to only 0.6% of the total execution time. And this overhead could be almost completely eliminated by simply compiling the L* code into machine code.[13]

This apparently large advantage for HSP in locking overhead must be qualified. HSII provides a more complete locking facility than does HSP, in that locks can extend over longer intervals of KS activity. (The effect of locking in HSP extends only over a single PS cycle). For example, in HSP a production may fire as a direct result of other productions having fired during earlier cycles, and it may take action based on an implicit condition (established by the earlier productions) that has since been invalidated. HSII does not completely solve this problem, but it does do better than HSP in so far as its larger knowledge units permit locking over longer time intervals.

However, HSII's more complete locking facility may be unnecessary. Lesser and Fennell [1977] feel that the basic self-correcting nature of HSII may allow it to tolerate a moderate amount of synchronization errors (i.e., actions based on partially invalid data), in the same way that it tolerates errors in input or inaccurate knowledge. If this is the case, then HSII can use a much simplified locking mechanism (just enough to prevent destruction of Blackboard structures). This would cause HSII's overhead to drop, but probably not as low as HSP's because HSII does not have the three properties discussed above. If, on the other hand, the more complete locking is essential, then HSP must have a more elaborate locking mechanism, resulting in sharply increased overhead.[14]

The HSII architecture needs a local context mechanism to maintain for each KS and PRE instantiation a local database of relevant changes to the Blackboard. The primitive Blackboard accessing routines are responsible for maintaining these local contexts based on a specification by each KS and PRE of what classes of changes it is interested in. Each PRE has a dynamic context which is continuously receiving records of relevant changes. When a PRE is instantiated, its static context gets a copy of the current dynamic context, and the dynamic context is cleared. When a KS instantiation is invoked, its local context gets a copy of both the static and dynamic contexts of the PRE that invoked it. When a KS terminates, its local context must be cleared. All this activity is classified as system

---

[12] Remember we are separating time lost due to blocking on a semaphore from the overhead involved whether a block occurs or not. The former is certainly not negligible, and is discussed in Section 5.5.

[13] This is a modest compilation task: the part of the interpreter that would have to be compiled has only about 130 symbols (in its PL* form).

[14] This would seem to require a fundamental architectural change to HSP, with explicit locking and unlocking operations for production actions, and with a delaying of evaluation for productions whose conditions access currently locked WM elements.

overhead, and Fennell and Lesser cite a figure of about 10%; but the figure is closer to 15% if we include the local context copying costs that they included under process handling overhead.

Part of the activity included in the local context overhead in HSII should probably not have been counted as such. Namely, the determination of which PREs to instantiate as a result of a Blackboard change, which is actually part of the evaluation of the precondition.[15] In HSP the production indexing (described in Chapter 4) has the analogous function of determining which productions to evaluate based on a WM change, and it seems obvious that this should not be counted as overhead. In HSII, this amounts to about 4% of overall runtime; thus, deducting it from local context overhead leaves us with 11% as the true overhead.

The HSP architecture avoids the need for any local context overheads. The global synchronization of the recognize-act cycle and the absence of any delay between condition and action remove the requirement for local contexts. Productions in HSP respond without delay to single WM changes. Thus, the local context of HSII is replaced in HSP by a single cell holding the change being responded to. And the "maintenance" of this Involves negligible overhead.

Another form of overhead in HSII is <u>process handling</u> -- the invocation, creation, and scheduling of PRE and KS processes. Fennell and Lesser give a figure of 9% overhead, but a significant portion of this (an estimated 5%) is local context manipulation associated with PRE and KS invocation, which we have classified under local context overhead instead.

Although the Fennell and Lesser system did simulate the scheduling of ready processes to processors, including descheduling and context swapping due to Blackboard interference, the associated overhead costs were not singled out.[16] It is known, however, that scheduling and context swapping costs are minor compared to the other process handling costs of creation, invocation, and cleanup.[17] In fact, this is why process handling overhead percentage does not vary much with different numbers of processors, even though the number of context swaps does vary from a few to over 900.

There is very little overhead in HSP for process handling. Process creation is done during initialization for an entire run, and thus creation overhead has been ignored in all the timing data presented.[18] Since processes are permanently and individually bound to processors, there is no overhead in scheduling processes to processors. And since all

---

[15] This activity has been called the <u>pre-precondition</u> in HSII circles.

[16] There is data on the <u>number</u> of context swaps, but no notion of how long a swap takes.

[17] Lesser, private communication, 1977.

[18] For the record, creation of a new L* process takes about 1.0 seconds plus 0.2 seconds for each local page which must be copied (HSP normally has two). The size of this number effectively prohibits any multiprocessing strategy involving dynamic process creation.

blocking on semaphores is done via busy-waiting (tight execution loops), there is no descheduling and context swapping going on.[19]

Access to the HSP change and production queues can be considered as overheads analogous to HSII process handling: insertion into the change queue as PRE invocation, insertion into the production queue as KS invocation, etc. In POM cycle 2, for example, the total overhead of this kind was calculated to be only about 1%. And most of it could be eliminated by compiling the HSP interpreter into machine code.

Figure 5.3 summarizes multiprocessing overheads for the HSII and HSP architectures. On this basis alone, HSP has a clear advantage: not only is execution overhead dramatically lower, but the system itself is simpler since it needs much less in the way of multiprocessing mechanism. However, it must be remembered that HSII is providing a more complete synchronization facility (which may or may not be necessary), and that this handicaps HSII in the comparison.

	% HSII	% HSP
Locking	27	1
Local context	11	0
Process handling	4	1
Total	42	2

Figure 5.3 Summary of multiprocessing overhead

5.5 Degree of Parallelism

Parallelism in HSII

The HSII system used by Fennell and Lesser for their multiprocessor simulation was at an early stage of development and thus did not have a full complement of KSs. In fact, many of its KSs were eliminated or replaced by the time of the final HSII system of September 1976. Fennell and Lesser's basic system had 6 PREs and 8 KSs, all operating at lower levels of speech knowledge: parametric, segmental, phonetic, surface-phonemic. The absence of KSs at higher levels (e.g., syllabic, word, and phrasal) means that their results underestimate the parallelism possible in a full HSII system.

[19] There is in fact no Hydra scheduling of HSP processes going on at all; HSP could run on a bare machine with no scheduler, assuming there were some way to get it initialized and started.

Figure 5.4 shows the effective parallelism achieved in a series of HSII simulations with increasing numbers of processors. In the 16 processor run the number of runnable processes never exceeded 16, so we can conclude that the maximum effective parallelism of this configuration is about 4.3. The reason for the slight decrease in effective parallelism from 8 to 16 processors is rather subtle, and is not important in this context.

# of processors	1	2	4	8	16
% processor utilization	99	98	95	54	26
Effective parallelism	0.99	1.96	3.80	4.32	4.16

Figure 5.4 Parallelism in the 8 KS, 6 PRE HSII configuration [Fennell and Lesser, 1977]

To explore the effect on parallelism of additional speech knowledge, Fennell and Lesser tested a second HSII configuration with a new PRE and KS operating at the phrasal and word levels of the Blackboard. As shown in Figure 5.5 below, it gave an effective parallelism with 16 processors of 5.28, up from 4.16. But it should be noted that this new KS was operating at Blackboard levels disjoint from all the other KSs, and thus there was no increase in Blackboard interference. In a more typical case a new KS would interfere with one or more existing ones, causing a diminished gain in parallelism.

Parallelism in the utterance-time dimension is affected strongly by the rate of data input at the lowest level of the Blackboard (in this case, the unclassified segments of the utterance). If the system were to truly operate in real time, parallelism in the utterance-time dimension would be essentially zero since the system would be forced to complete the processing on each unit of data before the next appeared. However, it is not unreasonable to assume instead that the entire input is available from the start (instantaneous input), thus permitting maximal parallelism in the utterance-time dimension. Fennell and Lesser did two simulation runs of the augmented configuration with 16 processors: one with left-to-right input and one with instantaneous input. The effective parallelism increased from 5.28 to only 5.6 with instantaneous input; the smallness of this increase is because of Blackboard access interference.

configuration	16 processors 8 KSs, 6 PREs l-to-r input	16 processors 9 KSs, 7 PREs l-to-r input	16 processors 9 KSs, 7 PREs instantaneous Input
% processor utilization	26	33	35
Effective parallelism	4.16	5.28	5.60

Figure 5.5 Parallelism in three different HSII configurations [Fennell and Lesser, 1977]

In the HSII simulations two sources of lost parallelism are distinguished: idle time when a processor has no process assigned and there are no ready processes to run, and lost time when a process is suspended and there are no ready processes to take its place on the processor. This lost time can be considered to be due solely to Blackboard interference, and as such the term blocked time will be preferred in what follows. Figure 5.6 shows idle and blocked time for the various HSII configurations. Note that each column adds to 100, except for rounding errors.

speech knowledge	8 KSs, 6 PREs					9 KSs, 7 PREs	
input form	I-to-r					I-to-r	instantaneous
# processors	1	2	4	8	16	16	16
% processor utilization	99	98	95	54	26	33	35
% idle time	1	1	2	14	46	36	34
% blocked time	0	0	2	32	28	31	32

Figure 5.6 Lost time in HSII: idle and blocked time [Fennell and Lesser, 1977]

Fennell and Lesser also made some simulation runs in which the locking mechanism was simply turned off, effectively eliminating Blackboard interference.[20] The configuration they used was: 9 KSs and 7 PREs, instantaneous input, and 32 processors. The effective parallelism achieved was 14.72. Compared with the value of 5.28, this shows that Blackboard interference has a major impact on parallelism.[21] This value of 14.72 represents an upper bound for what could be achieved with better locking and scheduling mechanisms that reduce interference. Or, as discussed above, it may be possible to simply do away with locking, in which the case the value of 14.72 would be realized.

Parallelism in HSP

The HSP POM system can be considered comparable to a 2 PRE, 2 KS HSII configuration: one PRE and KS for POM itself, and another pair for RPOL (although RPOL is small compared to POM). Thus the Fennell and Lesser HSII system had about a factor of 4 more knowledge content, including a mixture of bottom-up and top-down activity, giving it perhaps 2 to 3 times the potential parallelism along that dimension. The HSP runs were done with instantaneous input. But since only a single syllable from an utterance was used, HSP gives away a factor of about 10 to 20 in potential parallelism along the utterance-time dimension.

[20] This could have resulted in garbage, but the interesting fact is that it seemed to make little difference to overall system behavior. This was largely the basis for Fennell and Lesser's [1977] speculation, mentioned earlier, that Blackboard locking could be eliminated.

[21] The difference between 5.28 and 14.72 is due to a combination of removing the interference loss (about 30%), removing the overhead (again, about 30%), plus doubling the number of processors from 16 to 32.

HSP POM was run on C.mmp with a varying number of processors, and the resulting data on effective parallelism is shown below in Figure 5.7 for three selected cycles and for the total run. Of all 21 cycles, Cycle 10 was typically the one with the lowest parallelism, Cycle 9 the highest, and Cycle 15 was intermediate.

# of processors	1	2	4	7	10
Total time (sec)					
Cycle 10	14.9	9.7	7.1	6.8	5.4
Cycle 15	47.5	24.9	13.4	9.3	7.4
Cycle 9	99.3	50.7	26.2	16.4	12.3
all cycles	917	492	281	199	173
% processor utilization					
Cycle 10	100	77	52	31	28
Cycle 15	100	96	89	73	64
Cycle 9	100	98	95	86	81
all cycles	100	93	82	66	53
Effective parallelism					
Cycle 10	1	1.5	2.1	2.2	2.8
Cycle 15	1	1.9	3.5	5.1	6.4
Cycle 9	1	2.0	3.8	6.1	8.1
all cycles	1	1.9	3.3	4.6	5.3

Figure 5.7 Parallelism in the real HSP POM runs

The HSP simulator was then used for runs with up to 50 processors, but only for cycles 15 (intermediate) and 9 (large), with results shown below in Figure 5.8. Note that the results for 10 processors do not agree exactly with the corresponding real run, but are within 5%, and that is as good as could be expected given the difficulties listed in Sections 5.2 and 5.3.

# of processors	1	10	20	30	40	50
Total time (sec)						
Cycle 15	48.31	7.65	5.26	4.64	4.64	4.64
Cycle 9	99.09	11.69	6.85	5.56	4.91	4.91
% processor utilization						
Cycle 15	100	63	46	35	26	21
Cycle 9	100	85	73	59	51	40
Effective parallelism						
Cycle 15	1	6.3	9.2	10.4	10.4	10.4
Cycle 9	1	8.5	14.5	17.8	20.2	20.2

Figure 5.8 Parallelism in many-processor HSP POM simulations

The results show that even in a configuration with somewhere between 20 and 60 times less potential parallelism than Fennell and Lesser's HSII, HSP has higher parallelism. With 7 or 8 processors the parallelism is roughly comparable (around 4 or 5), but in HSII it bottoms out soon after that, while in HSP it continues to rise as more processors are used. The HSP simulations show that for a small cycle the parallelism levels off at about 8 between 20 and 30 processors, while with a large cycle it reaches 20 between 30 and 40 processors.[22]

As in the preceeding section on overhead, this HSII-HSP comparison must be qualified because of the fact that HSII locking is more complete. HSII is handicapped with lost time due to Blackboard access interference and locking overhead. HSP has no comparable losses since it operates without explicit locking of the WM. If the HSII locking could be safely removed, HSII parallelism would increase by roughly a factor of 3 (at least for the configuration used by Fennell and Lesser). If, on the other hand, explicit WM locking had to be added to HSP, its parallelism would surely fall (how much is unknown). In any event, we expect that HSP would maintain a significant edge (at least a half order of magnitude) over HSII in parallelism since HSP exploits intra-KS parallelism as an additional source.

The degree of parallelism in HSP is affected dramatically by the PM indexing mechanism. If no such mechanism existed, and hence every production in the system had to be evaluated for every WM change, the parallelism observed would be enormous due to all the unsuccessful production evaluations. But in a sense that would be cheating; the high parallelism would not be significant in such a grossly inefficient system. The point is that

[22] It was too costly to run the simulator on all cycles to get data for a total run.

HSP has a reasonable level of basic efficiency,[23] and yet has a high parallelism.

Since processor utilization during the action phase is determined solely by the number of processors, it is instructive to factor that out to see parallelism for the recognition phase alone. Figure 5.9 shows this both for the real runs and simulations.

	real runs					simulations				
# of processors	1	2	4	7	10	10	20	30	40	50
% processor utilization										
Cycle 10	100	99+	99	90	78	-	-	-	-	-
Cycle 15	100	99+	99+	92	85	83	72	60	45	36
Cycle 9	100	99+	99	97	93	97	93	82	74	59
all cycles	100	99	96	87	82	-	-	-	-	-
Effective parallelism										
Cycle 10	1	1.9+	3.9+	6.3	7.8	-	-	-	-	-
Cycle 15	1	1.9+	3.9+	6.4	8.5	8.3	14.4	17.9	17.9	17.9
Cycle 9	1	1.9+	3.9+	6.8	9.3	9.6	18.6	24.7	29.5	29.5
all cycles	1	1.9+	3.8	6.1	8.2	-	-	-	-	-

Figure 5.9 Parallelism in recognition phase of HSP POM

It was not possible, within the scope of this thesis effort, to get a good estimate of HSP parallelism for a full KS configuration operating on a complete input utterance. But we can get a rough picture as follows. We expect a 10 to 20-fold increase in utterance length to cause a similar increase in the size of each cycle, and little or no increase in the number of cycles (assuming instantaneous input). A 5 to 10-fold increase in the number of KSs would probably produce increases in both cycle size (2 to 3) and number of cycles (3 to 4). (The latter because of sequential dependencies between some KSs). The combined effect on cycle size would be a 20 to 60-fold increase.

Cycle size is the prime determinant of parallelism during recognition. Cycle 15 is

[23] There is evidence (mentioned in Chapter 4) to support this claim. HSP efficiency is comparable to two other PSAs: OPS and PSNLST.

about 3 or 4 times the size of Cycle 10,[24] with an increase in utilization from 78% to 85%; and Cycle 9, which is 2 or 3 times larger than Cycle 15, brings utilization from 85% up to 93%. A 20 to 60-fold increase in cycle size would bring the smallest cycle in the POM run up to twice the size of Cycle 9 (the largest). Thus we expect a utilization of close to 100% during recognition, at least for 10 processors and probably for even larger numbers. This being the case, the determinant of overall parallelism becomes the recognition-action ratio and the efficiency of the action phase (see below).

In HSP, lost parallelism can be grouped into idle time and blocked time, as with HSII, although they are not really comparable since they stem from different causes in the two systems. For example, HSII blocked time comes from the explicit synchronization of Blackboard access; HSP does no explicit synchronization, but the implicit synchronization of the PS cycle is a cause of idle time (during both recognition and action). As noted earlier this HSP system has only about one-third the knowledge content of the HSII simulation configurations. This makes detailed HSII-HSP comparisons inappropriate. However, the sparser knowledge content of HSP operates at fewer information levels, so that activity per level is not that different in the two systems. Figure 5.10 shows the data on lost time from the real HSP runs. The totals should be 100, but fall short for the 7 and 10 processor runs. This is likely due to hardware memory interference, which existed in small amounts in spite of special efforts to eliminate it (see following section).

# processors	1	2	4	7	10
% processor util.	100	94	82	66	53
% idle (total)	0	5	16	27	34
recognition	0	.4	1	3	5
action	0	6	15	24	29
% blocked (total)	0	.2	1	3	7
action interp.	0	.2	1	3	7
queue access	0	.0	.01	.02	.1
Total	100	99	99	96	94

Figure 5.10 Lost processor time in the real HSP POM runs

[24] Depending on how it is measured. One useful measure is (# of productions evaluated) + (# fired) + (# new changes). This yields a Cycle 15/10 ratio of 146/27 = 5.4 and a Cycle 9/15 ratio of 384/146 = 2.6. A more exacting measure might weight these factors differently according to how much each contributes on average to execution time. Note that the actual execution time ratios (recognition only) are: Cycle 15/10 = 4.3 and Cycle 9/15 = 2.1.

The main source of idle time (and of lost time as a whole) is the sequential nature of action execution in each cycle; i.e., all processors but one are idle while that one performs all the WM changes one at a time. The amount of such action-idle time depends directly on the recognize-act ratio, which is 15 overall, but varies from under 3 to 54 across individual cycles. These ratios are not as high as would be expected for a more complete, more knowledge-rich PS, so action-idle time in HSP may be artificially high.[25] Action-idle time also rises rapidly with the number of processors, for obvious reasons. The simulation data shows that action-idle time for Cycle 15 rises from 24% of processor time with 10 processors to 43% with 50.

Such a serious drain on utilization leads one to think of system modifications. Two possibilities are described here. The first is a simple modification in the way WM changes are stored. Currently they are encoded as linked list structures in the same manner as WM elements. Adopting the more rigid encoding of a vector (with 2 to 5 elements, depending on the type of change) would allow changes to be stored in a fixed array. Then the act of deleting all changes from the previous cycle, which currently takes 40% of action time, could be reduced almost to nothing (just the resetting of an array index). This would increase utilization in the POM run with 10 processors from 53% to 61%. A second possible modification is to actually execute the changes in parallel during the action phase (but still requiring that action execution not begin until recognition is complete). Some synchronization would be necessary during parallel action execution, to preserve the integrity of list fields of WM elements, and to control modifications to the WM index structure. This synchronization would be a minor part of the total change execution, so would not seriously inhibit parallelism. The gain from parallel action execution is roughly estimated for the 10 processor POM run to be an additional increase in utilization from 61% to around 75%.

A second source of idle time in HSP is a consequence of the global synchronization of the recognize-act cycle: near the end of the recognition phase of each cycle the number of productions left to execute falls below the number of processors, leaving some idle. This recognition-idle time is a factor of 5 or 6 smaller than the action-idle time, but its relative importance would increase if optimizations of the action phase as outlined above were incorporated. Fortunately, increasing cycle size works in favor of reducing the percentage of recognition-idle time, so that more KSs and longer input utterances will balance somewhat the negative effect of more processors. For example, Cycle 9 is about 2 or 3 times as large as Cycle 15, and has only 1% recognition-idle time compared to 4% for Cycle 15.

The principal source of blocked time in HSP is the critical section surrounding action interpretation. This relatively large amount of interference comes more from the particular implementation than from the HSP architecture. The architecture dictates only that representations of changes and WM elements be accessible by all the processors. The current HSP implementation puts these in a single shared memory area with a mutual

[25] One expects that the knowledge in HSP would become more complete primarily through the addition of more specialized knowledge. And "more specialized" implies higher recognition-action ratios, both through more complex production conditions, and a larger number of productions which must be evaluated to get the same number of firings.

exclusion semaphore for creation of changes and new WM elements within action interpretation. One improvement on this is to reduce interference by dividing the memory space for changes into several independently locked pieces. An even better improvement comes as a side effect of the action phase optimization discussed above that encodes changes as rows in an array. This would dramatically decrease creation time for a change representations by eliminating several list cell allocations. And since change creation time is currently a major cost in action interpretation, the size of the critical section would also be significantly reduced.

A second source of blocked time, interference in accessing the production and change queues, is insignificant for up to 10 processors. And the simulations indicate that it remains insignificant (less than 1%) for up to 50 processors.[26] Although optimization of the queue-accessing critical sections is currently unnecessary, they could be 20 times faster if written in machine code rather than interpreted L* code. Such an optimization could be resorted to if a great many processors were to be used.

5.6 Hardware Memory Interference

The problem of hardware interference due to multiple processors accessing shared memory is a crucial one for the current version of HSP running on C.mmp. The main reason for this is not some feature of the PSA of HSP, but rather the interpretive character of L* (the implementation language).[27] This section is included here largely for general interest.

It is a fact of life on C.mmp that 3 or 4 PDP-11/40 processors can saturate a single memory port. The processor-memory crosspoint switch adds some overhead to the memory cycle time; but worse, it clusters a number of independent memory modules (each with 8K, or 2 pages) into a single memory port. For example, suppose four processors are accessing pages in four different memory modules. Presumably, memory concurrency across the modules would mean that there is no interference. But if all the memories happen to be connected to the same switch port, concurrency across modules helps little, and serious interference results.

Given this fact of the underlying hardware, it would seem crucial for a user of the system to have control over the placement of his program and data pages in primary memory. Unfortunately, this is not the case.[28] However, a feature was added to Hydra to

[26] Although the simulator cannot predict queue access interference accurately, it suffices for this upper bound of 1%.

[27] There is a minor connection, in that PSAs seem to require interpreters of some sort in their implementation, and thus will always exhibit some degree of lopsidedness in their memory access pattern.

[28] The original design for C.mmp included a cache for each processor. This would presumably eliminate the interference problem.

allow a user to find out after the fact which memory port any given page is in. This allowed a strategy of trial-and-error to hit upon a distribution of pages across memory ports that would result in negligible interference.

Any system with a reasonably unconcentrated pattern of memory accesses would probably not experience serious degradation on C.mmp due to hardware interference. Unfortunately, HSP is close to a worst case. It is constructed as a double-layered interpretive system, as shown in Figure 5.11; the production language is interpreted by routines written in the L* language, which is itself interpreted by machine code routines. Thus a majority of memory accesses must be to the L* kernel (i.e., the interpreter, primitives, etc.). Further, most of the remaining memory accesses must be to the HSP kernel (production interpreter, etc.). It can be expected that only a very small fraction of accesses will be to the real code and data of the system (i.e., productions, WM elements, and the WM and PM indexes).

```
Productions
     |
     | Production interpreter
     |
L* routines
     |
     | L* interpreter
     |
Machine code
     |
     | Hardware of machine
```

Figure 5.11 Layers of interpretation in HSP

To investigate the pattern of memory accesses in HSP, some data was gathered with the Hardware Monitor.[29] A small (about 10 second) section from the condition evaluation phase of an HSP test run was monitored to obtain a profile of memory accesses across the entire user address space. The monitor sampled instructions at a rate of about 1% for a total of 56,000 sampled instructions. Figure 5.12 shows the results. (Note that page slot 0 is the first 4K portion of address space, and so on). A complete breakdown into L* kernel vs. HSP kernel vs. productions and WM elements was not possible because data access for the L* and HSP kernels are lumped together. But it can be seen that SHRPG1 (slot 1), which contains most of the L* kernel machine code, received 68% of the memory accesses.[30] And considerably less than .3% of accesses were to productions, WM elements, and the WM and PM indexes.

[29] The Hardware Monitor is a special hardware device, supported by its own PDP11/45, with monitoring probes connected directly to C.mmp hardware. It can be used to make a great variety of measurements. See [Marathe, 1977] for further details.

[30] An interesting side note is that about 70% of those (and thus 50% of the total) were to the L* interpreter, which is only a little over 100 words long.

page slot	%	page	shared or local	contents
0	12	Stack page	local	L* execution stack and operand stack
1	68	SHRPG1	shared	L* kernel code (incl. most machine code)
2	1.5	SHRPG2	shared	L* kernel code and data
3	17	LOCPG	local	L* kernel data and HSP kernel data
4	1	HSPPG	shared	HSP kernel code (incl. production interpreters)
		SYSPG	shared	Overlayed L* code
5	< .1	WMPG	shared	HSP data (incl. WM index)
		SYS2PG	shared	Overlayed L* code (small amount)
6	< .1	HSPU6PG	shared	Overlayed HSP kernel code and utilities
		all PM pages	shared	HSP productions
		PMXPG	shared	HSP PM index and related kernel code
7	< .1	HSPU7PG	shared	Overlayed HSP kernel code and utilities
		all WM pages	shared	HSP WM elements
		PMX pages	shared.	HSP continued PM index

Figure 5.12 Percentage of memory accesses in HSP by page slot

This data suggested that memory interference would indeed be a serious problem as long as SHRPG1 was shared among processors. Partly by a stroke of good luck, SHRPG1 contains nothing that has to be shared, so it was possible to turn it into a local page to give each process its own copy. From the data in Figure 5.12 it can be seen that once SHRPG1 is made local, a staggering 97% of memory references are to local pages. Thus, if C.mmp could be run with each processor's three local pages in a separate memory port, only 3% of each processor's memory accesses could possibly interfere with other processors, and memory interference would be essentially non-existent. In practice this ideal binding of pages to ports was not attainable. But with a little trial-and-error it was possible to keep more than two separate copies of SHRPG1 from ending up in the same port,[31] and this was sufficient to keep interference at a negligible level.[32]

To demonstrate the seriousness of the interference problem, HSP runs on C.mmp were

[31] In one run six attempts were necessary.

[32] The data given below in Figure 5.13 show that two copies of SHRPG1 sharing a port gives only about 4% degradation.

made without the individual copies of SHRPG1. Figure 5.13 shows the results. Two processors sharing SHRPG1 show only a 4% degradation, but this rises to over 100% with 10 processors. Note that somewhere between 7 and 10 processors, additional processors actually begin to increase total time. Also note that 10 interfering processes are considerably less effective than 4 non-interfering ones.

# processors	1	2	4	7	10
Time (sec)					
With SHRPG1 copied	917	492	281	199	173
Without SHRPG1 copied	917	513	412	321	353
Factor of degradation due to interference	1	1.04	1.47	1.61	2.04

Figure 5.13 Degradation of HSP execution time due to hardware memory interference

But what effect would optimization of HSP have on this strategy of copying code pages? Removing the L* interpretation level by compiling the HSP interpreter into machine code would result in a higher percentage of accesses to HSP shared data, productions, and WM structures (currently totalling less than .3%). But the remaining level of HSP interpretation would still produce a situation with references to the HSP kernel code being roughly 10 times as frequent as references to productions or WM elements.[33] There would still be much to gain by copying code pages.

Summary

No separate summary for this chapter appears here. See the section on parallelism in the conclusion chapter (Chapter 7).

[33] This estimate comes from the data of Figure 5.12, together with the observation that compiling the HSP kernel routines would almost certainly increase their size (and thus the number of accesses necessary to execute them). Note that the role of HSP kernel data accesses in this hypothetical situation cannot be estimated since they are mixed with L* kernel data accesses in the data of the figure.

Chapter 6

The Small Address Problem

A single C.mmp processor, being a 16-bit PDP11, can only access 64K bytes in its address space. But ample amounts of primary memory are available on C.mmp, around two million bytes. This mismatch leads to the problem of how to obtain access to more memory than fits in the address space, and we call this the small address problem. As discussed in [Wulf and Harbison, 1978], the hardware designers were pretty much forced into this problem, as it seems to be inherent in any multiprocessor built from small machines. It was originally hoped that large systems could be decomposed into a large number of processes each addressing only a small amount of memory, but experience on C.mmp has not borne this out except in exceptional cases.

To circumvent the small address problem, Hydra and the C.mmp hardware support a partitioning of every processor's address space into 8 slots, each of which holds one page of 8K bytes. Relocation registers provide a fast way of swapping these pages in and out of the slots in the address space. The time required for a swap of one of these pages, including operating system overhead, is only 20 microseconds.[1] This fast swap time makes possible various overlaying schemes that would be infeasible on a more typical computer system forced to swap to secondary memory.[2]

The C.mmp version of the implementation system L* [Newell, McCracken and Robertson, 1977] provides some support for dealing with the small address problem.[3] Within L*, the four page slots which make up the upper half of a processor's address space can be used for swapping pages.[4] The current version (D) of L*C.mmp binds every overlay page in a system to a fixed slot.[5] Overlays defined as lists of pages (at most one page for each of the four available slots) can be inserted into the address space and then later removed (returning the previously addressable pages). This overlaying operation can be used not

[1] This is a microcoded function of the operating system. The previous non-microcoded version took 200 microseconds.

[2] Use of the relocation registers is analogous in some respects to "base register" usage on 360-370 style machines.

[3] A general Hydra address-space-management facility has been designed [Hydra group, 1977], but there is as yet no experience with its use. Several large non-L* systems that have been built on C.mmp/Hydra have implemented their own specialized overlaying mechanisms. The Hydra kernel itself is a prime example of this.

[4] Of the lower four page slots, Hydra claims one and L* the other three -- they must not be overlayed.

[5] Version A, which allowed pages to float into any unused slot, ran afoul of the resulting complications and was abandoned.

only for large, modular subsystems, but also at the level of individual routines. In fact, a sizeable number of the routines in the basic L* system exist in overlays, and so must be swapped in when executed and then back out. Another overlay operation is provided to swap in an overlay without first preserving addressability of the overlayed pages (I.e., a replace operation rather than an push). This is useful for iterating through the multiple pages of a large data structure addressed through a single slot.

The cost in execution time of these overlaying mechanisms is not severe. In a run of the HSP POM+RPOL configuration, only 3% of execution time was taken by overlay swapping.[6] A second form of execution overhead necessitated by the overlay structures is time to do temporary copying of data structures into the fixed portion of the address space[7] to preserve their addressability during conflicting overlaying operations. This overhead amounted to only another 4 to 9% in the above-mentioned run. Thus execution overhead is not the problem.

The real essence of the small address problem is what it does to design, coding and debugging times. Working with a system requiring more than just a few overlay pages becomes quite complex. Correctness of page accessibility assumptions must be painstakingly maintained. The absolute page boundaries cause severe allocation problems for large, growing data structures. And some attention must always be given to limiting the amount of overlay swapping (though, as noted above, that is less important).

The debugging difficulties stem from this: if, while accessing data on an overlay page, a routine makes a mistaken assumption about which pages are currently addressable, there is fundamentally no way to detect the error.[8] The result will be a totally unpredictable error, and the symptom may not appear until long after occurrence of the error itself. Such errors are notoriously difficult to find, and they occur with dismaying frequency, at least until the programmer has learned to be careful about such things.[9]

The small address problem is serious, perhaps even crippling, for many large systems. It thus comes as good news that the HSP architecture provides some help.[10] The basic idea is this: WM elements are spread across multiple pages to be overlayed in one slot, and productions in multiple pages In another slot. Then once the small address problem has

[6] This was without the microcoded operating system function, but that matters little since overhead within L* dominated.

[7] The lower four pages of the address space are called fixed since they cannot be overlayed.

[8] If the data item is accessed via an L* external name, the error will be caught at "compile-time" since external name dictionaries and overlays are tied together. But many data accesses in the course of program execution are not via external names.

[9] At what cost in extra mental load?

[10] HSII was too large to fit on its PDP10, and this problem was solved by overlaying single KSs (and small groups) from a secondary storage device (drum). The HSII architecture thus helped solve that problem, but such a solution would not work on C.mmp since the KSs are too large to fit at once into the address space.

been solved for the HSP kernel,[11] the system can be grown indefinitely (barring other limits) without further concern. The characteristics of the HSP architecture that permit this are: (1) the existence of all long-term knowledge in the form of small, self-contained units (productions), and (2) the existence of a single global working memory, again with small units. (WM elements do reference each other, but this is handled by a special "large address" data type which identifies a page in addition to an address). Theoretically, the address space slots for productions and WM elements need only be large enough for a single unit, but larger slots reduce overlay swapping.[12] A large slot is particularly important for the WM elements since many of them are accessed for every production evaluated. In HSP a page slot can hold roughly 50 to 200 productions or 15 WM elements.

Since a key part of HSP's solution to the small address problem is a distribution of large, dynamic structures across multiple pages, there are formidable allocation problems. HSP has so far adopted simple and inflexible solutions. For example, the way in which the PM index is spread across pages is dependent on prior programmer knowledge of the total set of productions to be included in the system. Also, the distribution of WM elements across pages is controlled via a conservative maximum number per page, ignoring the actual amount of space used. The situation of running out of space on some page during creation of a WM element cannot be handled. These and similar problems must be solved before HSP can be considered adequate for general use as a PSA. Their solution, though difficult, seems straightforward.

[11] A substantial task in itself given its size and the size of the underlying L* system.

[12] The use of condition and action procedures also argues for larger slots since a procedure and all productions using it must be concurrently addressable.

Chapter 7

Conclusion

The central goal of this thesis is to evaluate the desirability of using a production system architecture (PSA) to implement a Hearsay-II-like speech understanding system. The approach of the thesis involved implementation of a PSA (called HSP) on the C.mmp/Hydra multiprocessor system, and translation of a body of HSII KSs into productions for HSP. The final evidence is rather ragged in several respects, but this was anticipated. The large size of the HSII-to-HSP KS translation effort, and the instability of C.mmp and its operating system Hydra are two of the main causes of this. Another is the inherent difficulty of comparing two systems as complex as HSII and HSP.

The promise of PSs for speech understanding was introduced via a list of characteristic implementation problems (see Figure 1.1). How has HSP fared with respect to the problems? Most of the evidence deals only with the first four problems: adequacy of representation, space efficiency, time efficiency, and the small address problem. No systematic study was made of the remaining six problems: error, directionality, augmentation, testing, debugging, and performance analysis. This does not reflect the relative importance of the problems. Rather it results mostly from time constraints. Also, the decision to maintain close comparability of HSP and HSII prevented free exploration in HSP of solutions to many of the problems.

The main body of this chapter is structured as a list of assertions based on the thesis research. Following the assertions, a short section brings together the various assertions for an overall conclusion taking their relative importance into account. Next comes a section discussing what the next steps should be if the evaluation of a PSA for HSII were to be continued. Several questions have emerged as the important ones still to be answered. A final section summarizes the contributions of the thesis.

7.1 The Assertions

The assertions are organized into five categories, roughly paralleling the division of the thesis into chapters. The exception is that representation and architecture are considered together here.[1] The assertions are stated in a strong form. A discussion

[1] This is appropriate because architectural issues have an impact on the representation of speech knowledge, even though the issues may have been settled without regard to this impact. For example, the use of explicit conditions on changes (discussed below as assertion 4) was motivated purely by architectural considerations. Yet it gives extra expressive power for knowledge representation.

following each assertion explains it and summarizes the supporting evidence. In cases where the evidence is weak, the assertion is qualified appropriately. Figure 7.1 gives a preview.

Representation and Architecture

1. HSP productions are adequate for representation of all HSII speech knowledge.
2. The adequate architecture of HSP is simple.
3. Translating HSII declarative knowledge lo HSP creates problems of multiple use.
4. Explicit conditionality on changes is a strong feature of HSP's architecture.

Space Efficiency

5. Procedural HSII knowledge decreases slightly in size when translated to HSP.
6. Declarative HSII knowledge increases in size by up to half an order of magnitude.
7. Total HSII knowledge increases in size by up to half an order of magnitude.
8. HSP requires a much larger global working memory than HSII.

Time Efficiency

9. The implemented time efficiency mechanisms in HSP are critical for its viabllity.
10. The time cost of HSP relative to HSII is at least two orders of magnitude.
11. Local working memory and control are the prime sources of HSII time efficiency.
12. Explicit pointers between WM elements in HSP greatly enhance time efficiency.

Parallelism

13. HSP exploits parallelism with lower overhead than HSII.
14. HSP achieves higher parallelism than HSII.
15. The limiting factor for HSP parallelism is sequential action execution.
16. Hardware memory interference does not seriously limit HSP parallelism.

Small Address Problem

17. The HSP architecture aids solution of the small address problem.

Figure 7.1 Assertions of the thesis

Before beginning with the assertions, we summarize briefly what the assertions are based on. For more information on the supporting evidence, consult the appropriate chapter in the body of the thesis.

Twelve HSII KSs were translated to HSP productions, and two of them were singled out for particularly thorough translation: POM, which recognizes syllables from phone-sized input segments;[2] and RPOL, which maintains the interrelationships of hypothesis validities. These translations provide the basis for comparisons of representation in HSII and HSP, notably for adequacy and space efficiency. The space efficiency comparisons are done separately for declarative and procedural HSII knowledge, since they behave differently under translation to HSP. Some of the comparisons are projected to include KSs that were not translated, on the basis of moderate understanding of how these untranslated KSs are structured.

Assertions about important features of the HSP architecture are supported by comparisons of HSP with three related PSAs: PSG [Newell, 1972], PSNLST [Rychener, 1976] and OPS [Forgy and McDermott, 1977], which represent an historical progression in development at CMU. These comparisons are also relevant to the assertions about time efficiency, since this is always a major concern for PSAs.

Equivalent HSII and HSP configurations containing only the POM and RPOL knowledge-sources were created and run on the same input (HSP being run in uniprocessor mode in this case). The data from these runs, once corrected for differences in the underlying systems,[3] support assertions about time efficiency.

Assertions about parallelism are based on multiprocessor runs of HSP (with the same POM + RPOL configuration), with up to 10 processors on C.mmp. Comparisons with HSII use data from the multiprocessor simulation of HSII by Fennell and Lesser, although their KS configuration is not directly comparable to HSP's. HSP parallelism is estimated for larger KS configurations, based on data from the existing configuration and knowledge of HSP's internal structure. A special HSP simulator was built and validated against the real multiprocessor runs. It collects data from uniprocessor runs and uses it to predict parallelism for larger numbers of processors (up to 50).

Now we begin with the assertions:

Representation and Architecture

1. *HSP productions are adequate for representation of all HSII speech knowledge.*

The basic evidence for adequacy[4] is that a large number (12) of HSII KSs were

[2] POM was chosen because it is one of the most complex and varied of all the HSII KSs. It has two intermediate levels of representation between input segments and syllables, and roughly a dozen processing stages, some simple and some quite complex in themselves.

[3] I.e., L* is slower than SAIL.

[4] By adequacy we do not mean theoretical adequacy (which is trivially present), but rather something we might call "representational practicality". We also do not mean to include time efficiency.

translated to HSP productions. There are nearly 20 other KSs which have been used at some time throughout HSII's development, but a moderate acquaintance with them has turned up no reasons to doubt HSP's adequacy. Efficiency is another matter (e.g., see assertion 11).

There were several minor difficulties encountered in the KS translation process, for example: iteration over WM element list fields, controlling duplicate actions, tallying events, and redundant arithmetic expression evaluation. Yet these difficulties are not serious enough to constitute a refutation of HSP's adequacy. Most of them could be solved through design iteration of the HSP architecture, without deviating from basic PS philosophy.

HSII KSs do a significant amount of simple table lookup from local arrays containing long-term knowledge. This might at first seem to cause a representation problem for HSP, but actually an array translates cleanly into a mutually exclusive set of productions, one for each entry. Direct indexing to select an array entry in HSII is essentially the same as selecting the correct production from the set in HSP. In other words, a PS is really a large, complex table lookup, while an array access is just an optimization of the lookup process that is possible with a highly uniform set of productions. However, a PSA is overly general for simple table lookup; the resulting space and time costs are discussed below under assertions 6 and 10.

The adequacy of HSP for speech understanding provides an additional data point for the broader question of adequacy of PSAs for artificial intelligence applications. More extensive evidence for adequacy has already been reported by Rychener [1976], in a study of six classical artificial intelligence programs. Although HSP and PSNLST (Rychener's PSA) differ significantly in their details, they are essentially the same when viewed from the world of artificial intelligence languages as a whole. Rychener's tasks span a large part of the domain of artificial intelligence: algebra problems, learning nonsense syllables, puzzles, chess endgames, natural language, and blocks manipulation. Yet the speech understanding task of HSP has some characteristics that set it apart: (1) a rich set of relatively independent knowledge sources, (2) each of which has a sound theoretical basis, providing lexicons of basic entities and rules relating entities, and (3) the large direct recognition component, i.e., no need for extensive serial reasoning. (Characteristic (2) is responsible for the large amount of knowledge which is conveniently represented in table form).

As for adequacy of PSAs in general for the speech understanding task, there can be little doubt. HSP is a simple PSA (see assertion 2), doing without several features that give an architecture more power. Further, the few idiosyncratic features that HSP does have do not affect power of expression so much as efficiency, which is beside the point for adequacy. Thus we can conclude that HSP's adequacy must extend to PSAs in general.

2. The adequate architecture of HSP is simple.

This assertion is of interest only in the context of assertion 1. Simplicity without assurance of adequacy is trivial.

Contrary to standard architectural practice for PSs, HSP has no conflict resolution; on each cycle it fires every true production. This permits a natural expression of the multiple, independent KSs that come from HSII and the speech task. Perhaps more importantly, it allows much greater parallelism: while PSG, PSNLST and OPS respond to only a few changes and fire only a single production each cycle, HSP can respond to hundreds of changes and fire a hundred productions or more. Such a feature could seriously impair a PS's ability to switch focus rapidly in response to a novel situation or new external stimulus.[5] But this does not seem worrisome for HSP since most of the multiple firings are operating in parallel on separate levels or time regions of the representation of the speech utterance.

The other side of the coin from responsiveness is stability: the ability of a PS to maintain a continuity of action over time. According to Rychener [1976] and McDermott and Forgy [1978], conflict resolution also plays an important role in stability. In fact, Rychener ranks event order (a conflict resolution principle which favors those production instantiations based on more recent WM changes) as the most essential feature of PSNLST, being used particularly for coordination and sequencing. In the absence of conflict resolution, HSP must occasionally resort to a somewhat inflexible mechanism called an n-cycle delay. This involves a chain of productions that waits for other activity to finish by marking time for a certain fixed number of cycles.

HSP permits neither disjunctions nor negated conjunctions within production conditions. (Negations of single condition elements are essential and of course permitted). These restrictions simplify the production interpreter without seriously affecting adequacy of representation. Disjunctions or negated conjunctions can be eliminated from a production by splitting it into several productions, at the cost of some decrease in space and time efficiency.[6] This splitting of disjunctions was frequently necessary in HSP, usually to accomplish iteration over a list field of a WM element. But this form of iteration is unnatural for a PS, having resulted from mimicking of HSII. Splitting of a negated conjunction was necessary in only a few cases, but the result was typically a complicated profusion of productions.

HSP has no mechanism for special case inhibition; i.e., preventing a true production from firing when another production representing a special case of the first one is also true. Such a mechanism would have complicated the production interpreter and probably caused a serious reduction in parallelism. Doing without special case inhibition is an inconvenience. Either the more general production must be augmented to make it complementary to the related special cases, or special productions must be added to detect when both general and special case productions fire and favor the special case result.

HSP's production language is at least half an order of magnitude simpler than HSII's language (a subset of SAIL). This conclusion is based on a comparison of primitive counts,

[5] See, for example, the importance McDermott and Forgy [1978] assign to such responsiveness, which they call sensitivity.

[6] The time cost is due to redundant evaluation of the same condition element in different productions. This cost is kept small by PSA efficiency mechanisms discussed under assertion 9.

number of data types, and sizes of runtime support and compilers. The extra complexity of SAIL permits efficiency of internal KS computation (see assertion 11).

3. Translating HSII declarative knowledge to HSP creates problems of multiple use.

A basic feature of the HSP architecture is the absence of any form of declarative long-term memory. Thus all long-term knowledge must be encoded as productions. Since each production specifies under what conditions its piece of knowledge is to apply, there is a problem with using that knowledge under different circumstances.[7] In some cases the problem can be solved by merely duplicating the knowledge, with a different production for each different use. This was done frequently in the HSP KSs, but will not extend to systems of growing complexity where multiple use can be expected to increase. Subroutines of productions provide another solution to the problem of multiple use. But subroutines require a high degree of similarity of the uses, plus rigid conventions for communication. A third, somewhat novel, solution employed in HSP is to deposit knowledge temporarily into WM whenever there is a reasonable expectation that it may be useful. In WM it is available in declarative form to whatever production wants to make use of it. Again, this implies conventions about how the knowledge is encoded in WM, plus a commonality in the variant conditions that can be used to trigger the depositing into WM. It is not known whether these solutions to the problem of multiple use would be adequate for much larger and more complex systems than HSII, but the suspicion is that they would not.

4. Explicit conditionality on changes is a strong feature of HSP's architecture.

The first condition element of every HSP production explicitly tests the nature of a WM change, and must match a change made in the previous cycle. Thus a production cannot fire any time its condition (excluding the first element) is true, but only when a particular type of change (tested by the first condition element) occurs in conjunction with a true condition. (Normally, the change <u>causes</u> the condition to become true). This explicit condition on a change has two important uses in HSP: (1) It provides the basis for the PM index described below under time efficiency. The PM index is an alternative to Forgy's scheme [1977] for reducing the dependence of execution time on PM size. (2) It solves the excitatory instability problem. A production cannot continue to fire cycle after cycle once become true, because it is also conditional on the occurrence of the change that made it true.

[7] This problem is also recognized by the Instructable Production System group [Rychener, Forgy, Langley, McDermott, Newell and Ramakrishna, 1977], but in the context of how to avoid repetition in instructing their PS about multiple uses. They propose general mapping mechanisms (in the form of more productions) to bridge the gap between variant uses.

Space Efficiency

5. Procedural HSII knowledge decreases slightly in size when translated to HSP.

Detailed space analysis of the POM KS shows that about 160 Kbits of long-term HSII procedural knowledge translated to only .7 times that much in HSP. There are two possible explanations for this decrease: (1) Since HSP productions are interpreted, a more compact representation is possible; and (2) HSP can represent condition testing and searching of the global working memory more concisely. However, the explanation is probably more complex than this, as suggested by high variation of the HSP/HSII space ratio. Eleven subparts into which POM was partitioned had ratios varying from .4 to 2.2, averaging to the .7 quoted above.

It was not possible to do space accounting of procedural knowledge for most of the 11 other translated KSs because the differences between the HSII and HSP versions are too great. However, crude data for one other KS, RPOL, shows an overall HSP/HSII space ratio of 1. Since HSII RPOL is virtually all procedural, this yields a procedural ratio of about 1, which is consistent with assertion 5 within limits of accuracy.

The procedural knowledge bit counts are supplemented by token counts[8] for 8 of the 11 POM subparts, which show an HSP/HSII ratio varying from .6 to 3.3, but averaging 1.0. Thus from a human perspective (assuming that humans perceive size by number of tokens) there is no change in the size of procedurally-encoded knowledge in translating from HSII to HSP.

6. Declarative HSII knowledge increases in size by up to half an order of magnitude.

HSII declarative structures are mostly either simple arrays, dictionaries of spellings, or linked networks, although all are encoded internally as arrays for efficiency. Arrays translate to HSP as one production per entry; the networks are typically represented by one production for each transition, or one production for all transitions out of each state. In any case, the number of productions required is large.

Seven instances of large declarative knowledge structures in the translated KSs were compared for size. Simple arrays require almost a factor of 4 more space in HSP, but sparseness can be exploited to reduce that factor by the fraction of non-default array entries. Tables of spellings (actually arrays of strings) have HSP/HSII space ratios of about 1. Two examples of network structures have the following ratios: 1.2 (grammar) and 3.2 (state transition network). The overall factor, excluding one exceptional case (a huge bit matrix), is 1.6.

HSP facilities for so-called condition and action procedures allow a sequence of similar condition or action elements to be packaged as a parameterized procedure and then

[8] A <u>token</u> is a lexical unit such as would be recognized by the language compiler.

referenced in many different productions with appropriate parameters supplied.[9] These facilities are crucial for representation of HSII declarative structures because of the large number of similar productions involved. In the HSP POM KS, a savings of a factor of 10 is obtained in the number of condition and action elements that must be represented explicitly. The bulk of this savings comes from productions encoding declarative HSII knowledge; the factor is much larger than 10 for these productions alone. Without procedures, assertion 6 might read "two orders of magnitude" or more.

7. Total HSII knowledge increases in size by up to half an order of magnitude.

In HSII POM, with an overall HSP/HSII space ratio of 1.1, the declarative/procedural split for long-term knowledge is .3/.7. But this is not a typical split. Many of the large KSs most recently added to HSII are estimated to be split about .9/.1. Assuming a .9/.1 split for a full HSII configuration would give an overall factor of 1.5, assuming the declarative HSP/HSII ratio to hold at 1.6. If instead we assume a declarative ratio of 3.8 (the worst observed), we get an overall factor of 3.5. We cannot be more precise than this because the declarative structures of the new KSs have not been analyzed. Their ratios may well be larger than 3.8.

Some related data comes from Rychener [1976] on PS translations of three classic artificial intelligence systems. Bit counts tend to be 2 or 3 times higher for the PSA, while very rough measures of source code size (related to our token counts) range from a slight disadvantage for the PSA to a factor of 3 or 4 advantage, depending on the comparison language. This bit count data agrees with assertion 7, even though Rychener's architecture is different from HSP, and his comparison systems different from HSII. This lends some credibility to the generality of assertion 7. However, this agreement could be mere coincidence; for example, HSP has to deal with data organized in tables, a disadvantage which does not seem to exist in Rychener's examples.

8. HSP requires a much larger global working memory than HSII.

Many HSII KSs use large local working memories in addition to the global Blackboard, and these local memories can be highly specialized for efficiency. HSP has only its global WM, and specialization of it is strongly limited by requirements of generality and uniform accessibility. This difference in specialization costs HSP less than half an order of magnitude for simple data items. For more complex data structures such as network nodes (as used in the SASS KS or the WIZARD procedure of the MOW KS), the cost can be as much as a full order of magnitude. Furthermore, such complex structures are so abundant that they dominate the overall cost.

[9] These procedure facilities are purely for space efficiency. A production using them behaves exactly as if all its elements were explicitly written out, except for a small time efficiency loss in assigning parameters to variables.

Time Efficiency

9. The implemented time efficiency mechanisms in HSP are critical for its viability.

Three existing time efficiency mechanisms in HSP give a combined speedup of 3 or 4 orders of magnitude over naive implementation (i.e., one that evaluates every production as a result of every WM change, and that searches WM for every condition element evaluated). All three mechanisms are suggested by analogs in the HSII architecture. See Section 4.1 for details.

> The first, called the PM index, reduces a linear dependence of execution time on PM size to sublinear by associating subsets of relevant productions (i.e., exactly those to evaluate) from classes of WM changes. In the POM run only 5 on average out of the 1000 total productions had to be evaluated per WM change, giving a speedup of about 200. In a larger system, say one with 10,000 productions, the number of productions evaluated per change could be expected to be not much greater than 5 (and certainly much less than 50). This is because new productions will often tend to be responding to changes different from existing productions.
>
> The second mechanism, use of explicit pointers between WM elements, reduces the amount of WM searching during condition element evaluation. In the POM run only one out of every 80 condition elements evaluated required WM searching; the remainder located the matching WM element directly via a pointer from some previously matched WM element. This mechanism does not reduce the degree of execution time dependence on WM size, but does give a constant factor of roughly 10 to 50 overall speedup in the POM run.
>
> The third mechanism, an index into WM according to the first two fields of a WM element, serves to reduce WM searching costs. Since it operates in the shadow of the second mechanism which greatly reduces the necessity for WM searching, the WM index makes only a small contribution.

These mechanisms bring HSP to roughly the same level of time efficiency as PSG with filters [McDermott, Newell and Moore, 1978], PSNLST, and OPS. Comparisons of the four PSAs were made, but on widely variant tasks, so details cannot be taken too seriously.[10] Underlying differences (machine, implementation language, etc.) were factored out as well as possible. Results of the comparison show the four PSAs within a half order of magnitude, or roughly equal in time efficiency considering accuracy of the comparison.

[10] It helps that each of the PSAs has efficiency mechanisms making it relatively insensitive to PM and WM size.

10. The time cost of HSP relative to HSII is at least two orders of magnitude.

In equivalent uniprocess runs of the POM + RPOL KS configuration, HSP took 255 times as long as HSII (917 sec as opposed to 3.6). This factor of 255 reduces to a range of 6 to 36 when corrected for eight underlying system differences: execution rate of machine, instruction set of machine, address space size, operating system, implementation system, degree of kernel optimization, speech knowledge, and complications of parallelism.

There are problems with the generality of this comparison since it is based on a single run of a small configuration (i.e., containing only the two KSs: POM and RPOL). The particular identity of the syllable input to POM is not important, but the syllable length is. The syllable used was the shortest possible; the longest possible syllable would increase the gap between HSII and HSP by another half to full order of magnitude. However, a factor of only about 2 is expected for a typical mix of syllable lengths. Relative to the smallness of the KS configuration, there are several important HSII KSs (SASS, WOSEQ, and MOW) which rely heavily on local control and working memory for efficiency. In their current form these KSs are seriously mismatched to the recognition-intensive HSP architecture, and would pay a time penalty of two orders of magnitude or more if directly translated. However, there may exist alternative formulations of these KSs that are better suited to HSP. We estimate that a full KS configuration would add another factor of one to one-and-a-half orders of magnitude over the POM + RPOL configuration. Putting together these corrections for syllable length and atypicality of the KSs gives a rough overall time efficiency loss of two to three-and-a-half orders of magnitude for a full configuration of KSs in HSP.

Related data by Rychener [1976] shows a factor of 6 to 10 loss in time efficiency for PS translations vs. original versions of six classic artificial intelligence systems. Rychener predicts that additional efficiency mechanisms in his PSA (e.g., compilation as proposed by Forgy [1977]) could get at least another factor of 5 improvement. Such a large similar improvement in HSP is not likely, due to existing mechanisms that reduce WM searching to a low level, but it is not unreasonable to expect an additional factor of 2 or 3. The large discrepancy of a couple of orders of magnitude between Rychener's six PSs and a full configuration of KSs in HSP is probably due to Rychener's tasks being better suited to the recognition-oriented character of a PSA than are the bulk of the speech KSs. The limited data for the POM + RPOL HSP configuration is in rough agreement with Rychener's data.

11. Local control and working memory are the prime sources of HSII time efficiency.

Since HSP has such limited local control (its actions are simple sequences and cannot call other productions), it must rely on data-directed invocation operating from the global WM. Much of the data-directed invocation might be avoided if productions could be much larger, compressing multiple PS cycles into single ones. But this is made difficult by an accompanying blowup in the number of productions.

The cost of data-directed control in HSP has several sources: The creation/deletion of change elements, creation/deletion of control signals in WM, PM indexing (finding productions to evaluate based on the changes), and an initial portion of condition evaluation for each production that is necessary to reobtain the context of the preceeding production. The total cost is estimated to be 30-45% of execution time.

HSP has data-directed knowledge units in the case of POM which are about 80 times smaller than HSII, and HSP utilizes 500 times as many data-directed invocations during execution. Yet the overhead is not 500 times greater; HSP data-directed invocations consume 30-45% of execution time as opposed to 9% in HSII. This is because the HSII overhead is dominated by the monitoring of Blackboard changes rather than by the actual invocations.

The other side of the coin from local control is local working memory use. HSII makes heavy use of local working memory for KS efficiency. Since a PSA contains no analogous facility, HSP is forced to use its global WM for such functions. The fact that HSP WM can be read without need of locking operations helps somewhat, but HSP is still at a serious disadvantage. The HSP run of POM + RPOL made over 5 times as many global working memory reads as a corresponding HSII run, and more than twice as many creations. Further, as mentioned above, there are other KSs that make much heavier use of local data. We expect HSP versions of these would make hundreds of times as many global working memory reads and creations as the HSII versions.

Converting HSP to local working memory use without local control makes no sense -- data-directed (global) control is based on the global WM. Thus the percentage of time spent in global WM access was estimated for a hypothetical HSP system already converted to use of local control (except for the few data-directed invocations necessary for inter-KS communication). The result is that 40% of execution time in such a system would be for WM access. And since corresponding local working memory operations are so much faster (50 to 600 times) in HSP than global ones, virtually all of that 40% could be eliminated.

Two other sources of HSP inefficiency are identified: the absence of a declarative long-term memory facility, and searching of WM. The former consumes 15% of execution time in the POM run (but some of this overlaps with the WM access costs discussed above). The latter is insignificant (3%) because of the explicit WM element references, but is projected to increase to around 30% of total time with a full input utterance.

Taking all the sources of inefficiency together, we can account for roughly a factor of 7 in execution time. This takes us well on the way toward explaining the normalized HSII-HSP difference of 6 to 36 obtained for the POM run. But it also suggests there may be other sources. One such possibility is the limited power of the HSP production language, as exemplified by the inability of a single production to deal with a data list of arbitrary length.

12. Explicit pointers between WM elements greatly enhance time efficiency.

The use of explicit pointers between WM elements, discussed above under time efficiency, provides an alternative to other known mechanisms for reducing the impact of WM searching on production evaluation time. One drawback of its use is a negative effect on representational issues (e.g., the possibility of spurious pointers to deleted elements); other common efficiency mechanisms are transparent to knowledge representation. The scheme used by OPS [Forgy, 1977] is relatively insensitive to the number of elements in WM. HSP's explicit pointers do not reduce the linear dependence of evaluation time on WM size, but do reduce the effect of WM size by a large constant factor (about 80 in the case studied), and thus will be most useful in applications for which WM size is small to moderate.

Parallelism

13. HSP exploits parallelism with lower overhead than HSII.

The HSP architecture requires only 2% multiprocessing overhead compared to 42% for Fennell and Lesser's multiprocessor version of HSII [Fennell, 1975; Fennell and Lesser, 1977]. HSP's low overhead is due to three factors: (1) the absence of explicit WM locking, (2) the absence of special local data contexts to hold relevant global working memory changes (as needed by KS instantiations in HSII), and (3) the simplicity of process handling. The absence of explicit WM locking was made possible by several features of HSP's PS architecture: the explicit separation of read activity (conditions) from write activity (actions), global synchronization of the recognize-act cycle, and sequential action execution (permitted by a high recognize-act ratio).

However, the 2% to 42% overhead comparison is unfair and must be qualified. HSII provides a more complete locking mechanism for global working memory, allowing locks to extend over longer intervals of KS activity. (In effect, HSP locking extends only over a single PS cycle). There are two possibilities: (1) The more stringent HSII locking is unnecessary. (Lesser and Fennell [1977] feel this may be so because of HSII's basic self-correcting nature which would allow it to recover from synchronization errors). Then HSII overhead could be much lower, but could probably not come within reach of HSP's 2%. (2) Stringent locking is necessary. In this case, HSP's architecture must be augmented to permit explicit WM locking, with the inevitable result of increased overhead. It is not known exactly how this augmentation might be done, nor how sharp the increase in overhead might be.

14. HSP achieves higher parallelism than HSII.

The degree of parallelism possible in HSP is higher than HSII. HSP's smaller knowledge unit size allows exploitation of <u>intra-KS</u> parallelism, both in the form of true parallel activity, and in parallel evaluation of conditions which turn out to be false. This intra-KS parallelism should account for at least a half order of magnitude Improvement In parallelism. The data obtained shows a much greater HSP-HSII margin than this, because HSII's more stringent locking handicaps it with overhead and lost time. Resolving the difference in locking between the two systems, while its exact effect is unknown, might wash away all of HSP's advantage except for some of the intra-KS parallelism.

HSP utilizes 53% of 10 processors in the POM run, which is roughly the same as that measured by Fennell and Lesser for an HSII configuration with 2 to 3 times the knowledge content and an input utterance 10 or 20 times as long (both of which increase the potential for parallelism). Furthermore, while HSII parallelism peaks by the time 10 processors are used, simulations of selected HSP cycles suggest that parallelism continues to increase with more processors, peaking between 30 and 40 processors. (The peak parallelism could not be precisely determined, but is probably between 7 and 10).

It was not possible to get a good estimate of HSP parallelism with knowledge content and utterance length comparable to HSII. But since cycle size is the prime determinant of parallelism during the recognition phase, and cycle sizes can be projected to be 20 to 60 times larger than in the POM run, we expect close to 100% utilization of 10 processors during recognition (and still high utilization with 50 to 100 processors). This means parallelism will be determined by the recognition-action ratio and the efficiency of action execution (see the following assertion).

15. The limiting factor for HSP parallelism is sequential action execution.

Lost time in HSP is dominated by the processor idle time caused by sequential action execution. For example, in the POM run with 10 processors, 29% of processor time was action-idle time, while other sources of lost time totalled only 12%. This action-idle time rises sharply with the number of processors: simulations of the POM run show a doubling of its percentage in going from 10 to 50 processors. High utilizations calculated for the recognition phase alone confirm the action phase as the limiting factor. With 10 processors, there is 82% utilization during recognition compared to 53% overall. And since recognition utilizations approaching 100% are projected for larger HSP configurations, the efficiency of action execution assumes a crucial importance.

There are a couple of possible optimizations that can be applied to action execution. One is to use a rigid, specialized format for change elements in order to reduce the overhead of change element deletion in the action phase.[11] The second is to make the changes in parallel. Some additional synchronization would be necessary to protect list fields of WM elements and the WM index structure; but this would not inhibit parallelism in a major way. It is estimated that the combined effect of these optimizations could raise utilization of 10 processors in the POM run from 53% to around 75%.

16. Hardware memory interference does not seriously limit HSP parallelism.

Measurement of memory accesses in an HSP run showed that considerably less than .3% of accesses were to shared HSP entities (productions, WM elements, etc.). Theoretically, this is so low that hundreds of processors could run in parallel without serious interference. In practice, however, there is also a substantial amount of shared code in the HSP kernel (production interpreters, etc.), resulting in 71% of all accesses being to shared pages. This creates a potential for serious interference.[12] One page

[11] This optimization would also decrease blocked time during the recognition phase since change element creation, which happens inside the action interpretation critical section, would also be significantly faster.

[12] Sharing of code is forced by a limit on the total number of pages in the system. When there are many processes, we can afford to copy only a few pages for every process.

containing L* kernel code (including the L* interpreters) accounted for 68% of accesses. By making multiple copies of this page, accesses to shared pages were reduced from 71% to 3%. This was sufficient to ensure negligible interference with 10 processors.

If HSP were optimized by compiling L* code into machine code, thus eliminating accesses to the L* interpreters, the majority of accesses would then be to the HSP kernel code (mainly the production interpreters). By making copies of HSP kernel code pages, the percentage of accesses to shared pages could again be brought down to a low level (though perhaps not as low as 3%).

Small Address Problem

17. The HSP architecture aids solution of the small address problem.

The problem of using a large primary memory through the window of a small address space is a serious and difficult one. This <u>small address problem</u> exists on C.mmp, with its million words of memory and its 16-bit PDP11 processors (with relocation registers), as it will on virtually any multi-mini or multi-micro-processor system. The HSP architecture aids the solution of this problem. Once the problem has been solved for the HSP kernel itself, a PS of any size may be accomodated without further concern.

A fast relocation-register load operation provided by the underlying C.mmp system is used by L* to provide overlay facilities. L* permits the upper half of the 32K address space to be overlayed by 4K pages in 4 page slots (sections of the address space). HSP uses one of these slots to access productions spread across multiple, mutually-exclusive pages, and another slot similarly for WM elements. The architectural features which permit this are the centralization of both long-term and short-term knowledge into memories (PM and WM) with small, self-contained units.

7.2 Summary of the Aspects Studied

We have shown that the HSP architecture is adequate for representing the HSII speech knowledge. This includes HSII declarative knowledge which must translate to procedural form in HSP. Adequacy of HSP was not a foregone conclusion because the simplicity of the HSP architecture compared to HSII gave grounds for some doubt about adequacy.

Space and time efficiency are another story. The moderate space penalty for representing declarative HSII knowledge in HSP is cause for concern since HSII does contain many large declarative knowledge structures. Even more serious concern arises over space inefficiency of the global HSP working memory, since it must be used in place of large, highly optimized local working memories that are typical in HSII KSs. HSP's lack of local working memory also causes a large loss of time efficiency because of greater creation/read/write costs and heavier use of data-directed control in global WM. HSP's large time efficiency handicap (two to three-and-a-half orders of magnitude) exists in spite of efficiency mechanisms which make HSP comparable in time efficiency to several other PSAs.

Some of the time efficiency handicap can be made up through increased parallelism. An additional source of parallelism, called intra-KS parallelism, results from HSP's smaller knowledge unit size. Intra-KS parallelism is due mostly to parallel evaluation of low-level alternative conditions rather than true parallel paths of computation within a KS (although the latter makes a small contribution). The effect on a full KS configuration is estimated conservatively to be a half order of magnilude increase in parallelism.

Except for the added parallelism, only one positive reason for using HSP has been revealed: help with the small address problem. Multi-micro and multi-mini processors such as C.mmp, for which the small address problem is an almost inevitable fact of life, are currently rare. But there is some chance they will become a dominant architecture. If so, solutions to the small address problem will be much in demand, and this could be a major impetus for use of PSAs. Other positive features of the HSP architecture may come from aspects not explored by this thesis. The following section identifies several candidates.

7.3 Questions for a Continued Evaluation

This section presents several important questions which have emerged from the current study. These questions are ones that need to be answered before the central question of the thesis can be finally resolved. Given the current state of the comparison, in which HSP fares poorly against HSII, some alleviation of inefficiencies or additional positive factors are sorely needed if the balance is to swing back toward HSP. Sources of such help may exist in the following list, but it is not possible to say for certain from our current state of understanding. Figure 7.2 gives a preview of the questions, after which each is remarked on briefly.

1. Can HSII KSs that heavily rely on local efficiency ultimately be made tractable in HSP?
2. Is the distributed control of directionality as dictated by a PSA feasible?
3. Can a significant improvement in ease of augmentation be realized in a PSA?
4. Do there or can there exist HSII KSs which cannot be represented adequately in HSP?
5. Do there exist situations that require the inhomogeneity of representation in HSP?
6. Is the level of individual productions right for performance analysis?
7. Do the simple language and small size of HSP productions make them more readable?
8. Can the HSP architecture be augmented with explicit WM locking if necessary?
9. Can a full HSP KS configuration productively use 50-100 processors?

Figure 7.2 Some remaining questions

1. Can HSII KSs that heavily rely on local efficiency ultimately be made tractable in HSP?

We have seen that the bulk of HSP's efficiency handicap relative to HSII comes from HSP's lack of local working memory and control. It is important to make some inroads on this handicap if HSP is to be useful, and several possibilities present themselves. First, optimization in the HSP kernel of the data-directed invocation and WM access mechanisms would not be difficult. But it would contribute only a modest improvement (perhaps a 50% overall speedup), which would not make a noticeable dent in the handicap of several orders of magnitude.

A second possibility, though one we would prefer not to use since it subverts some of the advantages of PSAs, is to permit productions to have complex actions with their own conditionality and local working memory. This is moving toward the large knowledge unit size of HSII. Perhaps there is some middle ground that recovers a large chunk of efficiency without sacrificing the benefits of a PSA (themselves poorly understood as yet).

There is a speculative possibility for regaining efficiency which entails breaking away from the close HSII-HSP correspondence purposely maintained in the thesis research. Perhaps the HSII KSs which do rely heavily on local efficiency could be restructured drastically to better suit the recognition-oriented PSA. There is a hint of a precedent for this in the history of HSII. Before the advent of the WIZARD procedure for word verification (one of the heavy users of local operations), there existed the WOMOS and POSSE KS modules which provided the same global function. These KSs were more recognition-oriented than WIZARD. Unfortunately, they were inherently much less efficient than WIZARD, so it is unlikely they would even be an improvement over WIZARD in HSP versions.

One final idea, again highly speculative, is to "compile" many PS cycles into one by combining production sequences into single large productions.[13] This would eliminate data-directed invocation and creation/deletion of control elements which mediate between the multiple cycles. In HSP this would cause a large increase in the number of productions, with high structural redundancy, but a compilation scheme such as OPS uses might solve this problem.

2. Is the distributed control of directionality as dictated by a PSA feasible?

Claims have been made that PSAs are unsuitable for representation of directionality control [Mostow and Hayes-Roth, 1978]. Thus it is incumbent upon us to prove them wrong if HSP is to survive its comparison with HSII. HSII has a specialized focussing mechanism which operates with global knowledge of the state of the computation. In HSP we must show (if possible) that the requisite global state information can be represented in WM, and continually updated by a set of added productions; and that task productions can be made to schedule themselves with additional condition elements sensitive to this representation of global state. If we cannot accomplish this, the conclusion that HSP is inadequate seems unavoidable. One difficulty that might be encountered is coordination of the global-state-maintaining productions with the task productions; the former are logically at a meta-level, yet are mixed in at the same level with the latter.

3. Can a significant improvement in ease of augmentation be realized in a PSA?

Ease of augmentation is one of the most promising aspects of a PSA, so we need to show a significant advantage here to balance negative aspects such as efficiency. One could propose case studies of augmentation in iterating HSP KSs. But a better strategy is to await results from related efforts such as the Instructable PS project [Rychener and Newell, 1978]. HSP's current architecture, lacking conflict resolution, may not properly support augmentation. If necessary, conflict resolution can be added to HSP, at the risk of sharply reduced parallelism.

[13] One difficulty we foresee is that the "n-cycle delay" mechanism used in HSP will prevent compilation of those cycles since the sequentiality is essential. There may be ways around this, perhaps by adding more power to HSP to make the n-cycle delays unnecessary. It is not clear that conflict resolution would work properly with these large productions.

4. Do there or can there exist HSII KSs which cannot be represented adequately in HSP?

Since there are about 20 HSII KSs which have not been translated to HSP, plus a virtually open-ended set of others which might conceivably be added to HSII, our evidence for adequacy is incomplete. Particularly suspicious are KSs such as WOSEQ and the new SASS, which use highly specialized local data structures. The current belief is that HSP is adequate for these, though inefficient. A more thorough HSII-HSP evaluation requires evidence for this belief.

5. Do there exist situations that require the inhomogeneity of representation in HSP?

The HSP architecture is better suited than HSII to representation of inhomogeneous knowledge, e.g., knowledge containing many special cases. HSII's strong point is homogeneous encodings; when it is forced out of these it must fall back on more expensive and ad hoc representations. This could turn into a strong advantage for HSP if the drive for improved performance pushed in the direction of inhomogeneity. This is uncertain at this point, but a weak argument can be made that as a system is extended and tuned it must necessarily incorporate more and more special case knowledge.

6. Is the level of individual productions right for performance analysis?

Performance analysis at the level of productions can be accomodated easily in HSP. The only issues are whether that level is a useful one (it is perhaps too low), and whether performance analysis at higher levels (always necessary) is made more difficult in a PSA. These issues can be resolved only by experience with larger KS configurations in HSP; the current two-KS configuration is too small to provide any significant insight. One possibility is that special-purpose productions can be added to handle the higher-level performance analysis, much in the same way that global state is to be maintained for directionality control.

7. Do the simple language and small size of HSP productions make them more readable?

The easier it is to characterize the knowledge content of a KS, the better (e.g., for communication with others in the scientific community). HSP productions seem to be much preferable to SAIL procedures: individual productions can be studied, and production counts are illuminating. But it is not so simple. There is the matter of control: productions can have complex interrelationships not at all apparent in their outward structure. Furthermore, much important HSII knowledge is in tables or networks, which are fine to read in their naked form. Obviously, more experience is required to answer this question.

8. Can the HSP architecture be augmented with explicit WM locking if necessary?

The point has been made that HSP's synchronization of WM access cannot currently extend over more than a single PS cycle. I.e., HSP has nothing like the ability in HSII of a KS obtaining arbitrarily long exclusive access to some part of global working memory. This more complete locking facility may turn out to be essential for HSP, so we must ask whether it can be added without destroying the architecture's positive features. Such an addition seems to require fundamental changes, such as explicit locking and unlocking operations as new action primitives, and possible delaying of evaluation for "blocked" productions. These changes seem feasible, but their success is by no means assured.

9. Can a full HSP KS configuration productively use 50-100 processors?

THe HSP simulator predicted moderate utilization of up to 50 processors with the two-KS configuration operating on a small fragment of input utterance. But there is a good prospect for much higher utilizations in a full KS configuration operating on a full utterance. This prospect makes a CM* [Swan, Fuller and Siewiorek, 1977] version of HSP attractive, but the poor overall performance of HSP relative to HSII makes such a version pointless for the time being.

7.4 Contributions

The contributions of this thesis come in three major categories:

First, of course, is the evaluation of a PSA for HSII -- the central goal of the research. A partial evaluation is presented, focussing on the problems of representational adequacy and space and time efficiency, which are incidentally some of the least promising for a PSA. From this vantage point, we ask questions that define appropriate directions for a continuation of the evaluation. Several of these directions hold promise for PSAs such as HSP. The partial evaluation is unfavorable for HSP, but at least we have quantified it so that a complete picture can be fit together as additional questions are answered.

We provide an enhanced understanding of HSII, a system which is a highly visible and important contribution to artificial intelligence, and thus worth knowing more about. By investigation through HSP of marked alternatives to HSII's philosophy, we shed light on it. For example, we contribute a better understanding of HSII's use of local memory and control for efficiency; we add force to the question of whether HSII could increase parallelism with greatly reduced or non-existent Blackboard synchronization, and possibly a decrease in knowledge unit size; and we call into question (though modestly) the strong belief that directionality control requires a separate, specialized mechanism.

We provide another data point for the applicability of PSAs to artificial intelligence systems. The significant aspect of the speech knowledge-sources that distinguishes speech understanding from the domains studied by Rychener [1976] is the heavy use of declarative knowledge structures.

We demonstrate several novel features which may be of use to designers of PSAs. We show that permitting multiple firings per PS cycle has appeal (at least for some tasks), and that eliminating conflict resolution does not necessarily cripple; that writing productions to contain explicit conditions on changes is a simple way to avoid repeated firing, and has efficiency benefits as well; that an attribute-value structure for WM and condition elements adds a useful bit of flexibility; and finally, that allowing WM elements to contain explicit references to each other dramatically cuts WM searching during evaluation.

Finally, we show that it is possible to obtain meaningful comparisons of related but different complex systems. The field of artificial intelligence could benefit from more comparisons of this sort.

References

Bernstein, M. (1976), "Interactive systems research: Final report", Technical Report TM-5243/006/08, System Development Corporation, Santa Monica.

Brooks, R. (1975), "A model of human cognitive behavior in writing code for computer programs", Ph.D. Thesis, Computer Science Department, Carnegie-Mellon University.

CMU Speech Group (1977), "Speech understanding systems: Summary of results of the five-year research effort", (second version), Technical Report, Computer Science Department, Carnegie-Mellon University.

CMU Speech Group (1976), "Working papers in speech recognition IV: The Hearsay-II system", Technical Report, Computer Science Department, Carnegie-Mellon University.

Cronk. R. and L. Erman (1976), "Word verification in the Hearsay-II speech understanding system", in [CMU Speech Group, 1976].

Davis, R. (1976), "Applications of meta-level knowledge to the construction, maintenance, and use of large knowledge bases", Ph.D. Thesis, Memo AIM-271, Artificial Intelligence Lab, Stanford University.

Davis, R. and J. King (1975), "An overview of production systems", Report STAN-CS-75-524, Computer Science Department, Stanford University.

Erman, L. and V. Lesser (1975), "A multi-level organization for problem solving using many diverse cooperating sources of knowledge", Proc. 4th IJCAI, pp.483-490.

Farley, A. (1974), "VIPS: A visual imagery and perception system; the result of a protocol analysis", Ph.D. Thesis, Computer Science Department, Carnegie-Mellon University.

Feigenbaum, E. A., B. G. Buchanan and J. Lederberg (1971), "On generality and problem solving: a case study using the DENDRAL program", Machine Intelligence 6, Edinburgh University Press.

Fennell, R. (1975), "Multiprocess software architecture for AI problem solving", Ph.D. Thesis, Computer Science Department, Carnegie-Mellon University.

Fennell, R. and V. Lesser (1977), "Parallelism in AI problem solving: a case study of Hearsay II", IEEE Transactions on Computers C-26, pp.98-111.

Forgy, C. (1978), "A production system monitor for parallel computers", Technical Report, Computer Science Department, Carnegie-Mellon University.

Forgy C. and J. McDermott (1977), "OPS: a domain-independent production system language", Proc. 5th IJCAI, pp.933-939.

Fuller, S.H. (1976), "Price/performance comparison of C.mmp and the PDP-10", IEEE/ACM Symposium on Computer Architecture, (Jan 1976), pp.195-202.

Hayes-Roth, F. (1973), "A structural approach to pattern learning and the acquisition of classificatory power", Proc. 1st Inter. Joint Conf. on Pattern Recognition.

Hayes-Roth, F., L. Erman and V. Lesser (1976), "Hypothesis validity ratings in the Hearsay-II speech understanding system", in [CMU Speech Group, 1976].

Hayes-Roth, F. and V. Lesser (1977), "Focus of attention in the Hearsay-II speech understanding system", Proc. 5th IJCAI, pp.27-35.

Hayes-Roth, F. and D.J. Mostow (1975), "An automatically compilable recognition network for structured patterns", Proc. 4th IJCAI, pp.246-251.

Hayes-Roth, F., D.J. Mostow and M. Fox (1977), "Understanding speech in the Hearsay-II system", in L. Bolc (ed.), Natural Language Communication with Computers, Springer-Verlag.

Hydra group (1977), "Hydra addressing specifications", unpublished.

Klahr, D. (1973), "A production system for counting, subitizing, and adding", in W.C. Chase (ed.), Visual Information Processing, Academic Press, pp.527-546.

Lesser, V. (1975), "Parallel processing in speech understanding: a survey of design problems", in [Reddy, 1975], pp.481-499.

Lesser, V. and L. Erman (1977), "A retrospective view of the Hearsay-II architecture", Proc. 5th IJCAI, pp.790-800.

Lesser, V. and R. Fennell (1977), "Parallelism in artificial intelligence problem solving", in [CMU Speech Group, 1977].

Lesser, V., R. Fennell, L. Erman and D.R. Reddy (1975), "Organization of the Hearsay-II speech understanding system", IEEE Trans. on Acoustics, Speech and Signal Processing, vol. ASSP-23, pp.11-23.

Lesser, V. and R. Suslick (1977), C.mmp version of Hearsay-II, private communication.

Lowerre, B. (1976), "The HARPY speech recognition system", Ph.D. Thesis, Computer Science Department, Carnegie-Mellon University.

Marathe, M. (1977) "Performance evaluation at the hardware architecture level and the operating system kernel level", Ph.D. Thesis, Computer Science Department, Carnegie-Mellon University.

McDermott, J. and C. Forgy (1978), "Production system conflict resolution strategies", in [Waterman and Hayes-Roth, 1978].

McDermott, J., A. Newell and J. Moore (1978), "The efficiency of certain production system implementations", in [Waterman and Hayes-Roth, 1978].

McKeown, D. (1977), "Word verification in the Hearsay-II speech understanding system", Proc. 1977 IEEE Conf. on ASSP, Hartford, pp.795-798.

Medress. M., F. Cooper, J. Forgie, C. Green, D. Klatt, E. Neuburg, A. Newell, M. O'Malley, D.R. Reddy, B. Ritea, J. Shoup-Hummel, D. Walker and W. Woods, "Speech understanding systems: Report of a steering committee", Sigart Newsletter No. 62, (Apr 1977), p.4-8.

Moore, J. and A. Newell (1974), "How can Merlin understand?", in L. Gregg (ed.), Knowledge and Cognition, Erlbaum.

Moran, T. (1973), "The symbolic imagery hypothesis: An empirical investigation via a production system simulation of human behavior in a visualization task", Ph.D. Thesis, Computer Science Department, Carnegie-Mellon University.

Mostow, D.J. and F. Hayes-Roth (1978), "A production system for speech understanding", in [Waterman and Hayes-Roth, 1978].

Newell, A. (1972), "A theoretical exploration of mechanisms for encoding the stimulus", in A.W. Melton and E. Martin (eds.), Coding Processes in Human Memory, Winston and Sons, pp.373-434.

Newell, A. (1973), "Production systems: models of control structures", in W. Chase (ed.), Visual Information Processing, Academic Press.

Newell, A. (1975), "A tutorial on speech understanding systems", in [Reddy, 1975], pp.3-54.

Newell. A., J. Barnett, J. Forgie, C. Green, D. Klatt, J.C.R. Licklider, J. Munson. D.R. Reddy and W.A. Woods (1973), Speech Understanding Systems: Final Report of a Study Group, North-Holland/American Elsevier.

Newell, A., D. McCracken and G. Robertson (1977), "L*: an interactive, symbolic implementation system", Technical Report, Computer Science Department, Carnegie-Mellon University.

Newell, A. and J. McDermott (1975), "PSG manual", Technical Report, Computer Science Department, Carnegie-Mellon University.

Newell, A. and H. A. Simon (1972), Human Problem Solving, Prentice-Hall.

Reddy, D.R. (ed.) (1975), Speech Recognition Invited Papers of the IEEE Symposium, Academic Press.

Reiser, J. (1976), "SAIL", Memo AIM-289, Artificial Intelligence Lab, Stanford University.

Rychener, M. R. (1976), "Production systems as a programming language for artificial intelligence applications", Ph.D. Thesis, Computer Science Department, Carnegie-Mellon University.

Rychener, M., C. Forgy, P. Langley, J. McDermott, A. Newell and K. Ramakrishna (1977), "Problems in building an instructable production system", Proc. 5th IJCAI, p.337.

Rychener, M. R. and A. Newell (1978), "An instructable production system: basic design issues", in [Waterman and Hayes-Roth, 1978].

Shortliffe, E.H. (1974), "MYCIN: A rule-based computer program for advising physicians regarding anti-microbial therapy selection", Ph.D. Thesis, Computer Science Department, Stanford University.

Smith, A.R. (1976), "Word hypothesization in the Hearsay-II speech understanding system", IEEE Int. Conf. on Acoustics, Speech and Signal Processing, pp.549-552.

Swan, R., S. Fuller and D. Siewiorek (1977), "Cm*: a modular, multi-microprocessor", AFIPS Conference Proceedings, Vol. 46, National Computer Conference, pp.637-644.

Walker, D. (ed.) (1976), "Speech understanding research: Final technical report", Stanford Research Institute.

Waterman, D. (1975), "Adaptive production systems", Proc. 4th IJCAI, pp.296-303.

Waterman, D. and F. Hayes-Roth (eds.) (1978), Pattern-Directed Inference Systems, Academic Press.

Winograd, T. (1972), Understanding Natural Language, Academic Press.

Winograd, T. (1975), "Frame representations and the declarative/procedural controversy", in D. Bobrow and A. Collins (eds.), Representation and Understanding, Academic Press, pp.185-210.

Woods, W. et al. (1976), "Speech understanding systems: Final technical report", BBN report 3438, Bolt Beranek and Newman, Inc., Cambridge.

Wulf, W.A. and C. G. Bell (1972), "C.mmp -- A multi-mini-processor", Proceedings of the Fall Joint Computer Conference, pp.765-777.

Wulf, W.A., E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson and F. Pollack, "Hydra: the kernel of a multiprocessor operating system", Comm. of the ACM, 17, 6 (June 1974), pp.337-345.

Wulf, W.A. and S. Harbison (eds.) (1978), "Reflections in a pool of processors: An experience report on C.mmp/Hydra", Technical Report, Computer Science Department, Carnegie-Mellon University.

Young, R. (1973), "Children's seriation behavior: A production-system analysis", Complex Information Processing No. 245, Department of Psychology, Carnegie-Mellon University. Also available from Department of Computer Science.

Appendix A

HSP Specifications

> Working memory (WM) is an unordered collection of WM elements (WMEs)

> There is no architectural limit to the size of WM

> WM elements

> A WM element (WME) is an association list structure of pairs: field identifier, field value
> E.g. < id1/val1 id2/val2 >
> These are very similar to Hayes-Roth's Parameterized Structural Representations (PSRs) [Hayes-Roth, 1973]

> The field identifier may be defaulted, in which case the field is identified positionally from the start of the WME (e.g. the third field value specified with a defaulted field identifier is taken as the third field from the start)

> Field values may be named symbols, integers, references to other WMEs, or single-level lists of the preceeding items

> Speech knowledge units (e.g., syllables, words) are defined in lexicons, which are themselves represented by symbols (e.g., SYL, WRD)
> The knowledge units are represented externally by a "..." notation, e.g., "BEEF" for the word 'beef'
> The most recent previous occurrence of a lexicon symbol determines which lexicon is to be used
> The external representation of a knowledge unit is translated internally to an integer, namely the index of the item in its lexicon

> There are two particularly common forms of WM elements: hyps (hypotheses) and links (support links between hypotheses)

> Hyps always begin with HYP as the first field value, the lexicon of the hyp second, and the knowledge unit symbol third (these fields are positionally

defined)
  > E.g. <HYP WRD "BEEF">
> Hyps also have a begin time field (which is a list of two integers: begin time and begin range) and an end time field (similar to begin time)
  > E.g. <HYP WRD "BEEF" BTIME/(20 3) ETIME/(60 0) >
> Hyps have an upper validity field (UVLD) which holds validity propagated upward from supporting hyps, a lower validity field (LVLD) which holds validity propagated downward, and a combined validity field (VLD); all validity fields have single integers between -100 and 100 as values
> Hyps also have an upper links (ULNKS) field which is a list of references to all the links which have the hyp as their lower hyp; and similarly, a list of alternative supporting links (LLNKS) which are links having the hyp as their upper hyp
> Links always begin with LNK as the first field value, the lexicon of the lower supporting hyp second, followed by a reference to the upper (supported) hyp, followed by a list of references to the sequence of lower (supporting) hyps, followed by both upper and lower implications (integers between -100 and 100 representing the upward and downward strength of the support)
  > E.g. <LNK WRD UHYP/H23 LHYPS/(H17 H18) UIMP/90 LIMP/30 > where H23 must be a word hyp

> Production memory (PM) is an unordered collection of all productions existing in the system

  > Productions consist of a sequence of condition elements (CEs) followed by the -> symbol, followed by a sequence of action elements (AEs)

  > A sublist of condition elements may take the place of a single element, to arbitrary depth; the sublist structure is transparent to the production interpreter
    > This transparent sublist facility is merely for allowing naming and/or sharing of CEs among productions

  > Condition elements

    > A CE is an association list structure closely paralleling the WME structure

> Field values of a CE may be the same as in WMEs, except that a variable or the special symbol * may also appear in place of any of the other possible items
> > By convention, any external symbol beginning with a $ is recognized as a variable
>
> A CE may also be the symbol NOT followed by an association list structure as above
>
> Or a CE may be a variable, followed by the symbol =, followed by an association list
> > Or a NOT followed by all three of these

> Condition element match

> > A CE is matched against a WME field for field from left to right for positional fields, otherwise by corresponding field identifiers
> > > Corresponding named symbols must satisfy a symbol equality test
> > > Corresponding integers must satisfy an integer equality test
> > > Corresponding references to WMEs must refer to the same WME
> > > Corresponding lists must match in every corresponding element (it is necessary that the lists be the same length)
> >
> > The first occurrence (in left to right order) of a variable in a condition indicates a variable binding is to occur, while successive occurrences of the same variable indicate a match against the variable's binding is to occur
> >
> > The special symbol * will successfully match any element it corresponds to
> >
> > The special symbol ** within a list will match zero or more items on the list
> > > Only one occurrence of ** is permitted in a list
> > > E.g. the variable **$LH** in **LHYPS/(* $LH **)** picks out the second lower hyp from the list if the list has two or more hyps
> > > E.g. **LHYPS/($LH1 ** $LH2)** will pick out the first and last if the list has at least two hyps
> >
> > A failure of the match at any point signals failure of the entire CE match
> >
> > Special match predicates may be composed from other

predicates with AND, OR and NOT enclosed within [ ]
- > The binary predicates (> and < for integers) must be written within [ ] to delineate the second argument
  - > E.g. [< 73]
- > The predicate = may be used inside [ ] to obtain the standard match operation for the particular field
  - > E.g. fldid/[=3] is the same as fldid/3 , but fldid/[3] will not work
- > NOT may be used within [ ] to reverse the sense of another predicate
  - > E.g. [NOT < 60] , [NOT = 6]
- > AND may be used to create conjunctive tests
  - > E.g. ETIME/([=$ET AND > $BT+12] 0) where the [...] applies to the first element of the ETIME list
- > OR may be used to create disjunctive tests
  - > E.g. fldid/[=1 OR =2 OR =3]
- > The binary infix arithmetic operators (+, -, *, /, MAX, MIN, ABS) may be used with integers and and integer variables within [ ]
  - > Inner levels of [ ] denote nesting of arithmetic expressions
  - > E.g. [< $BR+3] , [> [$V*$I]/100]

> Condition and action procedures

- > As mentioned above, CEs can be grouped into lists and named, and then used in multiple productions without duplicating the space
  - > These might be called parameterless procedures

- > A facility also exists for condition procedures with parameters
  - > This permits much greater sharing across productions since the use in each sharing production need not be identical, but only the same modulo a set of parameters
- > The definition format is:
  procname: (CPROC ($var1 $var2 ...) ce1 ce2 ce3 ...)
  where ce1 ce2 ce3 ... are CEs containing instances of the variables $var1 $var2 ...
- > The format of a call is simply:
  parm1 parm2 ... procname in a condition

- > A similar facility exists for actions using APROC,

except that a balancing APROC. is also needed
> E.g. procname: (APROC ($var1 ...) ae1 ae2 ... APROC.)

> Condition match

> By convention the first CE in a production refers to a symbolization of a change which must have occurred in the previous cycle of the PS
> E.g. <MOD H23 VLD> -- a change to the validity of hyp H23

> A backtracking mechanism in the production interpreter attempts all possible matches of CEs to WMEs, and the action will be executed once for each successful match
> Thus a single production may fire multiple times in a single cycle

> Each successive CE is matched either against the entire WM, or if the CE is of the form $var = < ... > and the variable $var was assigned in a previous CE of the same condition, the CE is matched only against the single WME (or list of WMEs) bound to $var

> When a CE contains a NOT, it will evaluate to true only if all possible matches against WMEs fail; thus there are no multiple satisfying assignments possible for a NOT CE
> It is meaningless to attempt to bind a variable within a NOT CE

> Action elements

> Action elements (AEs) are simple calls to one of the action primitives listed below

> WMEs and subelements which were bound to variables during the condition match are designated to the AEs by passing the variables as parameters

> AEs are written in the same way as WMEs or CEs using < >, giving a form of prefix function notation

> All the AEs produce as a side effect a symbolization of the change
> This is used to drive the following PS cycle

> <NEW <...> > or <NEW $wme=<...> >

- > Creates a new WME as specified by the attribute-value structure <...> (and binds it to variable $wme if present)
- > <DEL $wme>
  - > Deletes the WME bound to variable $wme
- > <MOD $wme fldid newval>
  - > Replaces the field specified by field identifier fldid of the WME bound to variable $wme with newval
  - > Will also create a new field in $wme if no field with identifier fldid already exists
- > <MOD.ADD $wme fldid $wmeref>
  - > Adds a reference to the WME bound to $wmeref to the beginning of the list of references that is field fldid of the WME bound to $wme
  - > Will also create a new field in $wme if no field with identifier fldid already exists
- > <MOD.ADDE $wme fldid $wmeref>
  - > Same as MOD.ADD except adds to the end of the list rather than the beginning
- > <MOD.REP $wme fldid $oldwme $newwme>
  - > Replaces a reference to $oldwme by a reference to $newwme wherever it occurs in the list that is field fldid of $wme
- > <MOD.DEL $wme fldid $wmeref>
  - > Similar to MOD.ADD except deletes the reference from the list
  - > Will also delete the field if the list becomes empty by deleting the reference
    - > Thus fldid/() should never be used in a condition, but rather NOT < ... fldid/* ... >

> Examples
- > <NEW $L= < LNK SYL UHYP/$W LHYPS/($M1 $M2) IMP/90 > >
  - > Creates a new link to a word hyp bound to $W, from the sequence of lower syllable hyps bound to $M1 and $M2, with implication = 90, and binds the new link to variable $L
- > <MOD $HL BTIME $BT>
  - > Replaces the current begin-time of hyp $HL (list of time integer and time range integer) with the time list bound to $BT
- > <MOD.DEL $LH ULNK $OL>
  - > Deletes the reference to link $OL from the ULNK field of hyp $LH

> The binary infix arithmetic operators (+, -, *, /, MAX, MIN, ABS) may be used within [ ] when necessary to compute an integer for new value
>
> > E.g. <MOD $LH LVLD [[$I*$LV]/100] >

> PS control
>
> > See Section 5.1 of the thesis (under "Parallelism in the HSP Control Cycle") for an explanation of global control in HSP

Appendix B

Sample HSP Productions

Below is an example of HSII declarative knowledge encoded as HSP productions -- a table of segment vowel probabilities. There is a separate production for each table entry (PD.VP1 through PD.VP43), though they all use the condition procedure POM.VPRB to save space (see the end of Section 3.1.4). (There are 43 altogether -- not all are shown below). POM.VPRB takes four parameters: the identity of the speech segment and its a priori vowel probability for each of three vowel types (A, I, U). (Note the order of the parameters is reversed from declaration to call). The productions trigger on a new segment hyp (hypothesis represented as a Working Memory element) and store the three probabilities as new fields of the hyp if the segment identity matches the one in the production. The Production Memory index has these productions indexed by four levels: NEW, HYP, SEG, and "xx" (the segment identity), so only the single correct production need be evaluated when a new segment hyp appears.

```
        POM.VPRB: (  CPROC ($UVP $IVP $AVP $X)
                     < NEW $S >
                     $S= < HYP SEG $X >
                         ->
                              < MOD $S AVPRB $AVP >
                              < MOD $S IVPRB $IVP >
                              < MOD $S UVPRB $UVP >  )

PD.VP1  :         ( "-"   0     0      0       POM.VPRB )
PD.VP2  :         ( "K"   0     0      0       POM.VPRB )
PD.VP3  :         ( "P"   0     0      0       POM.VPRB )
  ...
PD.VP32 :         ( "EY"  16    80     0       POM.VPRB )
PD.VP33 :         ( "EH"  30    39     17      POM.VPRB )
PD.VP34 :         ( "IH"  16    50     25      POM.VPRB )
PD.VP35 :         ( "IX"  0     100    0       POM.VPRB )
PD.VP36 :         ( "IY"  0     48     29      POM.VPRB )
PD.VP37 :         ( "EL"  15    0      48      POM.VPRB )
PD.VP38 :         ( "ER"  11    20     33      POM.VPRB )
PD.VP39 :         ( "OW"  24    11     58      POM.VPRB )
PD.VP40 :         ( "UH"  21    30     33      POM.VPRB )
PD.VP41 :         ( "UW"  0     0      49      POM.VPRB )
PD.VP42 :         ( "UX"  0     0      0       POM.VPRB )
PD.VP43 :         ( "AY"  47    39     0       POM.VPRB )
```

The three productions below (PD.CVPRB1, 2 and 3) calculate the combined vowel probabilities of each type (A, I, U) for an "option-seg". An option-seg is a hyp representing the combination of one to a maximum of three (by convention) alternative segments in the same time interval. The cases of there being one, two or three alternative segments are handled separately by the three productions. (This can be considered a representational weakness of HSP. An HSII KS can encode this much more concisely since it can handle an arbitrary number of alternatives with a single procedure). The parameterless condition procedures VPRB.OS and VPRB.S1, 2 and 3 are used just to save space. Note the complex arithmetic expressions within the actions to compute the combined vowel probabilities based on each alternative segment's vowel probabilities and normalized rating.

```
VPRB.OS : (     < MOD $S [ =NRAT OR =AVPRB OR =IVPRB OR =UVPRB ] >
                $S= < HYP SEG OSEGLNK/$OS >  )

VPRB.S1 : ( $S1= < AVPRB/$AVP1 IVPRB/$IVP1 UVPRB/$UVP1 NRAT/$NR1 >  )
VPRB.S2 : ( $S2= < AVPRB/$AVP2 IVPRB/$IVP2 UVPRB/$UVP2 NRAT/$NR2 >  )
VPRB.S3 : ( $S3= < AVPRB/$AVP3 IVPRB/$IVP3 UVPRB/$UVP3 NRAT/$NR3 >  )

PD.CVPRB1 : (   VPRB.OS
                $OS= < SEGS/($S1) >
                VPRB.S1
                    ->
                        < MOD $OS AVPRB $AVP1 >
                        < MOD $OS IVPRB $IVP1 >
                        < MOD $OS UVPRB $UVP1 >  )
PD.CVPRB2 : (   VPRB.OS
                $OS= < SEGS/($S1 $S2) >
                VPRB.S1
                VPRB.S2
                    ->
        < MOD $OS AVPRB [ [ [$AVP1*$NR1] + [$AVP2*$NR2] ]/100 ] >
        < MOD $OS IVPRB [ [ [$IVP1*$NR1] + [$IVP2*$NR2] ]/100 ] >
        < MOD $OS UVPRB [ [ [$UVP1*$NR1] + [$UVP2*$NR2] ]/100 ] > )
PD.CVPRB3 : (   VPRB.OS
                $OS= < SEGS/($S1 $S2 $S3) >
                VPRB.S1
                VPRB.S2
                VPRB.S3
                    ->
< MOD $OS AVPRB [ [ [$AVP1*$NR1] + [$AVP2*$NR2] + [$AVP3*$NR3] ]/100 ] >
< MOD $OS IVPRB [ [ [$IVP1*$NR1] + [$IVP2*$NR2] + [$IVP3*$NR3] ]/100 ] >
< MOD $OS UVPRB [ [ [$UVP1*$NR1] + [$UVP2*$NR2] + [$UVP3*$NR3] ]/100 ] >
                )
```

Below there are three productions (PD.GAP, PD.GAPL and PD.GAPR) which propagate segment "gaps" (missing segments represented as segment hyps with identity "-") up through the syllable and word levels (as "GAP" hyps at those levels). The three productions trigger when the option-seg hyp for a segment gap appears in Working Memory. The condition procedure NEWGAP merely tests for such a situation and sets up the context into some variables.

For the interior of an utterance, PD.GAP applies, under condition that the gap must be at least 15 time units long. PD.GAPL and PD.GAPR permit a gap shorter than 15 to be propagated if it abuts the utterance-begin segment ("[") or utterance-end segment ("]"). Note that PD.GAPL and PD.GAPR must ensure that the segment is indeed shorter than 15, else there might be a duplicate firing of one of them and PD.GAP. This is an example of extra encoding that is necessary to prevent duplicate firings in the absence of a conflict resolution mechanism in the architecture.

```
NEWGAP : (        < NEW $OS >
                  $OS= < HYP OSEG SEGS/($S) >
                  $S= < HYP SEG "-" BTIME/($BT *) ETIME/($ET *) >  )


NEWGAPS: ( -> < NEW $M= < HYP SYL "GAP" BTIME/($BT 3) ETIME/($ET 3) > >
                  < NEW < LNK OSEG UHYP/$M LHYPS/($OS) UIMP/100 > >
                  < NEW $W= <HYP WRD "GAP" BTIME/($BT 3) ETIME/($ET 3)
                            IVLD/100 SYLNUM/1 > >
                  < NEW < LNK SYL UHYP/$W LHYPS/($M) UIMP/100 > >  )


PD.GAP : (        NEWGAP
                  $S= < ETIME/( [ > $BT+15] *) >
                      ->
                          NEWGAPS )

NEWGAP.E : (      NEWGAP
                  $S= < ETIME/( [ NOT > $BT+15] *) >  )

PD.GAPL : (       NEWGAP.E
                  < HYP SEG "[" ETIME/($BT *) >
                  NOT < HYP WRD "GAP" BTIME/($BT *) ETIME/($ET *) >
                      ->
                          NEWGAPS  )
PD.GAPR : (       NEWGAP.E
                  < HYP SEG "]" BTIME/($ET *) >
                  NOT < HYP WRD "GAP" BTIME/($BT *) ETIME/($ET *) >
                      ->
                          NEWGAPS  )
```

The productions below cooperate to idenlify those option-segs which have high enough combined vowel probabilities to act as syllable "nuclei". There are two classes of nuclei (NUCs): strong and weak; and those option-segs that are neither must be marked accordingly (for later reference). This happens in three consecutive PS cycles:

First, PD.SNUC will mark as NUC/STRONG an option-seg that satisfies the strong criteria (vowel probability greater than .3 and existence of an amplitude maximum, or MXN hyp, which overlaps the option-seg). At the same time (i.e., also in the first cycle), PD.FSNUC initiates an explicit 1-cycle delay by creating a special WM element <POM FSNUC option-seg> which will cause triggering on the following cycle. (Note this element also records the relevant option-seg hyp).

On the following cycle, PD.WNUC reacts to the special element and tests whether the option-seg was marked STRONG on the previous cycle. If not, and if it satisfies the weak criteria, it is marked as NUC/WEAK. In the meantime, PD.FWNUC creates another 1-cycle delay with the element <POM FWNUC option-seg>.

On the third cycle, PD.NONUC reacts to the new <POM FWNUC option-seg> element and tests whether the option segment has been marked as either STRONG or WEAK, and if not marks it as NONUC/TRUE.

PD.FNUC. is the cleanup production. It deletes both kinds of special delay elements in the cycle after which they were created.

PD.ANUCL and PD.ANUCR monitor an "error" condition; namely the appearance of two contiguous (in time) NUCs. Their actions (encoded in the procedure ANUC.AOS) are actually L* code (the underlying implementation language) which prints a diagnostic message. This should be considered an expedient rather than part of the HSP architecture.

```
PD.SNUC : (      < MOD $OS VPRB >
                 $OS= < HYP OSEG VPRB/[ > 30 ] BTIME/($BT 0)
                                                ETIME/($ET 0) >
                 < HYP MXN BTIME/([ < $ET] 0) ETIME/([ > $BT] 0) >
                     ->
                         < MOD $OS NUC STRONG >  )

PD.FSNUC : (     < MOD $OS VPRB >
                 $OS= < HYP OSEG >
                     ->
                         < NEW < POM FSNUC $OS > >  )

PD.WNUC : (      < NEW $X >
                 $X= < POM FSNUC $OS >
                 NOT $OS= < NUC/* >
                 $OS= < VPRB/[ =$VP AND > 10] BTIME/($BT 0)
```

```
                       ETIME/( [ =$ET AND > $BT+12] 0) >
                   NOT < HYP OSEG ETIME/($BT 0) VPRB/[ NOT < $VP] >
                   NOT < HYP OSEG BTIME/($ET 0) VPRB/[ NOT < $VP] >
                   NOT < HYP OSEG ETIME/($BT 0) NUC/* >
                   NOT < HYP OSEG BTIME/($ET 0) NUC/* >
                        ->
                             < MOD $OS NUC WEAK >   )

PD.FWNUC : (       < NEW $X >
                   $X= < POM FSNUC $OS >
                        ->
                             < NEW < POM FWNUC $OS > >   )

PD.NONUC : (       < NEW $X >
                   $X= < POM FWNUC $OS >
                   NOT $OS= < NUC/* >
                        ->
                             < MOD $OS NONUC TRUE >   )

PD.FNUC. : (       < NEW $X >
                   $X= < POM [ =FSNUC OR =FWNUC ] >
                        ->
                             < DEL $X >   )

ANUC.OS : (        < MOD $OS NUC [ =STRONG OR =WEAK] >
                   $OS= < HYP OSEG BTIME/($BT 0) ETIME/($ET 0) >   )

ANUC.AOS : (       $AOS= < NUC/[ =STRONG OR =WEAK] >
                        ->
                          S"Adj NUCs: " WRHSP $OS PRHSP& SPACE2 $AOS PRHSP )

PD.ANUCL : (       ANUC.OS
                   $AOS= < HYP OSEG ETIME/($BT *) >
                   ANUC.AOS  )

PD.ANUCR : (       ANUC.OS
                   $AOS= < HYP OSEG BTIME/($ET *) >
                   ANUC.AOS  )
```